λSnap!

Build Your Own Blocks

4.0

SNAP! Reference Manual

Brian Harvey

Jens Mönig

Table of Contents

Acknowledgements

Snap! Reference Manual
Version 4.0

Snap! (formerly BYOB) is an extended reimplementation of Scratch (**http://scratch.mit.edu**) that allows you to Build Your Own Blocks. It also features first class lists, first class procedures, and continuations. These added capabilities make it suitable for a serious introduction to computer science for high school or college students.

To run Snap!, open a browser window and connect to either **http://snap.berkeley.edu/run** to start with a minimal set of blocks or **http://snap.berkeley.edu/init** to load a small set of additional blocks (a little slower startup, but recommended for convenience and assumed in this manual).

I. Blocks, Scripts, and Sprites

This chapter describes the Snap! features inherited from Scratch; experienced Scratch users can skip it.

Snap! is a programming language — a notation in which you can tell a computer what you want it to do. Unlike most programming languages, though, Snap! is a *visual* language; instead of writing a program using the keyboard, the Snap! programmer uses the same drag-and-drop interface familiar to computer users.

Start Snap!. You should see the following arrangement of regions in the window:
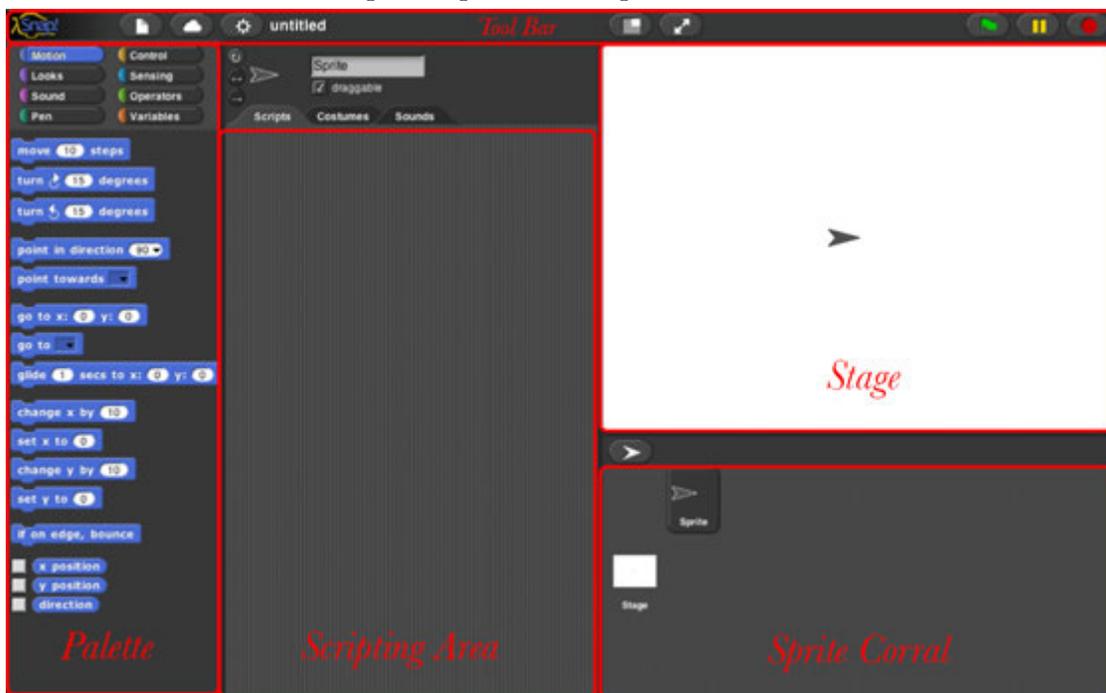
(The proportions of these areas may be different, depending on the size and shape of your browser window.)

A Snap! program consists of one or more *scripts,* each of which is made of *blocks.* Here's a typical script:
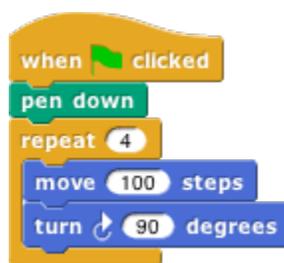
The five blocks that make up this script have three different colors, corresponding to three of the eight *palettes* in which blocks can be found. The palette area at the left edge of the window shows one palette at a time, chosen with the eight buttons just above the palette area. In this script, the gold blocks are from the Control palette; the green block is from the Pen palette; and the blue blocks are from the Motion palette. A script is assembled by dragging blocks from a palette into the *scripting area* in the middle part of the window. Blocks snap together (hence the name Snap! for the language) when you drag a block so that its indentation is near the tab of the one above it:
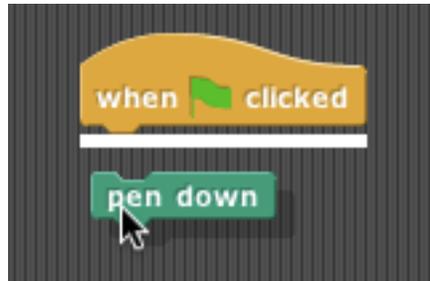
The white horizontal line is a signal that if you let go of the green block it will snap into the gold one.

Hat Blocks and Command Blocks

At the top of the script is a *hat* block, which indicates when the script should be carried out. Hat block names typically start with the word "**when**"; in this example, the script should be run when the green flag near the right end of the Snap! tool bar is clicked. (The Snap! tool bar is part of the Snap! window, not the same as the browser's or operating system's menu bar.) A script isn't required to have a hat block, but if not, then the script will be run only if the user clicks on the script itself. A script can't have more than one hat block, and the hat block can be used only at the top of the script; its distinctive shape is meant to remind you of that.

The other blocks in this script are *command* blocks. Each command block corresponds to an action that Snap! already knows how to carry out. For example, the block move 10 steps tells the sprite (the arrowhead shape on the *stage* at the right end of the window) to move ten steps (a step is a very small unit of distance) in the direction in which the arrowhead is pointing. We'll see shortly that there can be more than one sprite, and that each sprite has its own scripts. Also, a sprite doesn't have to look like an arrowhead, but can have any picture as a *costume*. The shape of the **move** block is meant to remind you of a Lego™ brick; a script is a stack of blocks. (The word "block" denotes both the graphical shape on the screen and the procedure, the action, that the block carries out.)

The number 10 in the **move** block above is called an *input* to the block. By clicking on the white oval, you can type any number in place of the 10. The sample script on the previous page uses 100 as the input value. We'll see later that inputs can have non-oval shapes that accept values other than numbers. We'll also see that you can compute input values, instead of typing a particular value into the oval. A block can have more than one input slot. For example, the **glide** block located about halfway down the Motion palette has three inputs.

Most command blocks have that brick shape, but some, like the **repeat** block in the sample script, are *C-shaped*. Most C-shaped blocks are found in the Control palette. The slot inside the C shape is a special kind of input slot that accepts a *script* as the input. In the sample script, the **repeat** block has two inputs: the number 4 and the script

A. Sprites and Parallelism

Just below the stage is the "new sprite" button . Click the button to add a new sprite to the stage. The new sprite will appear in a random position on the stage, facing in a random direction, with a random color.

Each sprite has its own scripts. To see the scripts for a particular sprite in the scripting area, click on the picture of that sprite in the *sprite corral* in the bottom right corner of the window. Try putting one of the following scripts in each sprite's scripting area:

When you click the green flag, you should see one sprite rotate while the other moves back and forth. This experiment illustrates the way different scripts can run in parallel. The turning and the moving happen together. Parallelism can be seen with multiple scripts of a single sprite also. Try this example:

When you press the space key, the sprite should turn forever in a circle, because the **move** and **turn** blocks are run in parallel. (To stop the program, click the red stop sign at the right end of the tool bar.)

Costumes and Sounds

To change the appearance of a sprite, import a new *costume* for it. There are two ways to do this. First, select the desired sprite in the sprite corral. Then, one way is to click on the file icon in the tool bar, then choose the "**Import…**" menu item. You can then select a file in any picture format (PNG, JPEG, etc.) supported by your browser. The second way is quicker if the file you want is visible on the desktop: Just drag the file onto the Snap! window. In either case, the scripting area will be replaced by something like this:

Just above this part of the window is a set of three tabs: Scripts, Costumes, and Sounds. You'll see that the Costumes tab is now selected. In this view, the sprite's *wardrobe,* you can choose whether the sprite should wear its Turtle costume or its Alonzo costume. (Alonzo, the Snap! mascot, is named after Alonzo Church, a mathematician who invented the idea of procedures as data, the most important way in which Snap! is different from Scratch.) You can give a sprite as many costumes as you like, and then choose which it will wear either by clicking in its wardrobe or by using the `switch to costume Turtle▼` or `next costume` block in a script. The Turtle costume is the only one that changes color to match a change in the sprite's pen color.

In addition to its costumes, a sprite can have *sounds;* the equivalent for sounds of the sprite's wardrobe is called its *jukebox.* Sound files can be imported in any format (WAV, OGG, MP3, etc.) supported by your browser. Two blocks accomplish the task of playing sounds. If you would like a script to continue running while the sound is playing, use the block `play sound Help!▼`. In contrast, you can use the `play sound Help!▼ until done` block to wait for the sound's completion before continuing the rest of the script.

Inter-Sprite Communication with Broadcast

Earlier we saw an example of two sprites moving at the same time. In a more interesting program, though, the sprites on stage will *interact* to tell a story, play a game, etc. Often one sprite will have to tell another sprite to run a script. Here's a simple example:

In the `broadcast bark▼ and wait` block, the word "bark" is just an arbitrary name I made up. When you click on the downward arrowhead in that input slot, one of the choices (the only choice, the first time) is "**new**," which then prompts you to enter a name for the new broadcast. When this block is run, the chosen message is sent to *every* sprite, which is why the block is called "broadcast." In this program, though, only one sprite has a script to run when that broadcast is sent, namely the dog. Because the boy's script uses **broadcast and wait** rather than just **broadcast**, the boy doesn't go on to his next **say** block until the dog's script finishes. That's why the two sprites take turns talking, instead of both talking at once.

Notice, by the way, that the **say** block's first input slot is rectangular rather than oval. This means the input can be any text string, not only a number. In the text input slots, a space character is shown as a brown dot, so that you can count the number of spaces between words, and in particular you can tell the difference between an empty slot and one containing spaces. The brown dots are *not* shown on the stage when the block is run.

The stage has its own scripting area. It can be selected by clicking on the Stage icon at the left of the sprite corral. Unlike a sprite, though, the stage can't move. Instead of costumes, it has *backgrounds:* pictures that fill the entire stage area. The sprites appear in front of the current background. In a complicated project, it's often convenient to use a script in the stage's scripting area as the overall director of the action.

B. Reporter Blocks and Expressions

So far, we've used two kinds of blocks: hat blocks and command blocks. Another kind is the *reporter* block, which has an oval shape: `x position`. It's called a "reporter" because when it's run, instead of carrying out an action, it reports a value that can be used as an input to another block. If you drag a reporter into the scripting area by itself and click on it, the value it reports will appear in a speech balloon next to the block:

When you drag a reporter block over another block's input slot, a white "halo" appears around that input slot, analogous to the white line that appears when snapping command blocks together. Here's a simple script that uses a reporter block:

Here the **x position** reporter provides the first input to the **say** block. (The sprite's X position is its horizontal position, how far left (negative values) or right (positive values) it is compared to the center of the stage. Similarly, the Y position is measured vertically, in steps above (positive) or below (negative) the center.)

You can do arithmetic using reporters in the Operators palette:

The **round** block rounds 35.3905… to 35, and the **+** block adds 100 to that. (By the way, the **round** block is in the Operators palette, just like **+**, but in this script it's a lighter color with black lettering because Snap! alternates light and dark versions of the palette colors when a block is nested inside another block from the same palette:
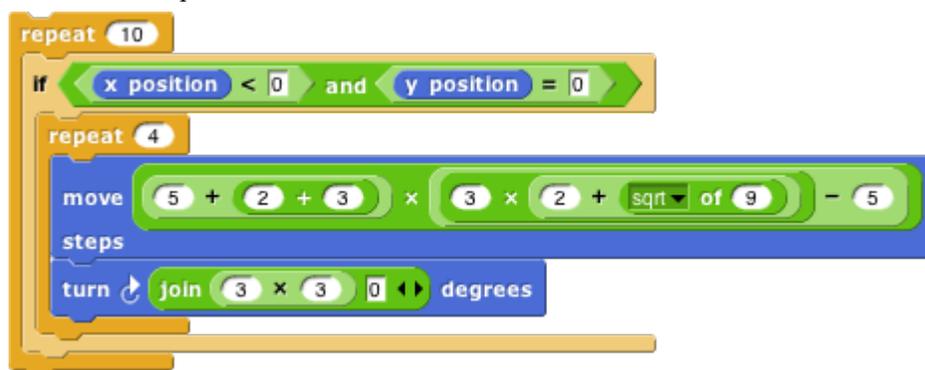
This aid to readability is called *zebra coloring.*) A reporter block with its inputs, maybe including other reporter blocks, such as `round x position + 100`, is called an *expression.*

C. Predicates and Conditional Evaluation

Most reporters report either a number, like `( ) + ( )`, or a text string, like `join hello· world ◀▶`. A *predicate* is a special kind of reporter that always reports **true** or **false**. Predicates have a hexagonal shape:

`‹ mouse down? ›`

The special shape is a reminder that predicates don't generally make sense in an input slot of blocks that are expecting a number or text. You wouldn't say `move ‹ mouse down? › steps`, although (as you can see from the picture) Snap! lets you do it if you really want. Instead, you normally use predicates in special hexagonal input slots like this one:

`if ‹ ⬡ ›`

The C-shaped **if** block runs its input script if (and only if) the expression in its hexagonal input reports **true**.

```
if ‹ (y position) < 0 ›
  say Help!·I'm·under·water!
```

A really useful block in animations runs its input script *repeatedly* until a predicate is satisfied:

```
point towards Sprite2 ▾
repeat until ‹ touching Sprite2 ▾ ? ›
  move 3 steps
```

If, while working on a project, you want to omit temporarily some commands in a script, but you don't want to forget where they belong, you can say

```
move 10 steps
turn ↻ 15 degrees
if ‹ false ›
  say I'm·not·going·to·do·this for 2 secs
  glide 1 secs to x: 0 y: 0
point in direction 90 ▾
```
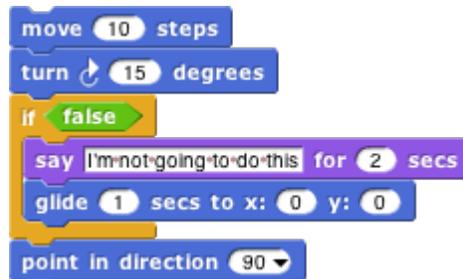
Sometimes you want to take the same action whether some condition is true or false, but with a different input value. For this purpose you can use the *reporter* **if** block:[1]

`say join I'm·on·the· ‹ if ‹ (x position) < 0 › then left else right › ◀▶`

The technical term for a true or false value is a "Boolean" value; it has a capital B because it's named after a person, George Boole, who developed the mathematical theory of Boolean values. Don't get confused; a hexagonal block is a *predicate*, but the value it reports is a *Boolean*.

Another quibble about vocabulary: Many programming languages reserve the name "procedure" for Commands (that carry out an action) and use the name "function" for Reporters and Predicates. In this manual, a *procedure* is any computational capability, including those that report values and those that don't. Commands, Reporters, and Predicates are all procedures. The words "a Procedure type" are shorthand for "Command type, Reporter type, or Predicate type."

---

[1] If you don't see it in the Control palette, click on the File button 🗋 in the Tool Bar and choose "**Import tools**."

D. Variables

  Try this script:[1]
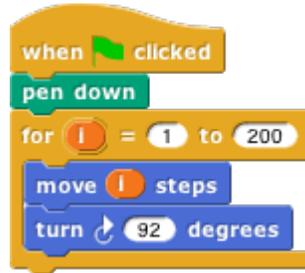
The input to the **move** block is an orange oval. To get it there, drag the orange oval that's part of the **for** block:
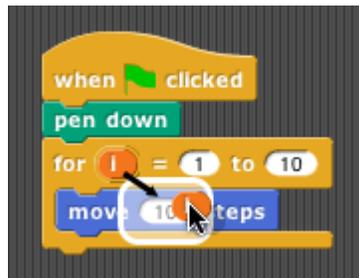
The orange oval is a *variable:* a symbol that represents a value. (I took this screenshot before changing the second number input to the **for** block from the default 10 to 200, and before dragging in a **turn** block.) **For** runs its script input repeatedly, just like **repeat**, but before each repetition it sets the variable **i** to a number starting with its first numeric input, adding 1 for each repetition, until it reaches the second numeric input. In this case, there will be 200 repetitions, first with **i**=1, then with **i**=2, then 3, and so on until **i**=200 for the final repetition. The result is that each **move** draws a longer and longer line segment, and that's why the picture you see is a kind of spiral. (If you try again with a turn of 90 degrees instead of 92, you'll see why this picture is called a "squiral.")

  The variable **i** is created by the **for** block, and it can only be used in the script inside the block's C-slot. (By the way, if you don't like the name **i**, you can change it by clicking on the orange oval without dragging it, which will pop up a dialog window in which you can enter a different name:

"**I**" isn't a very descriptive name; you might prefer "**length**" to indicate its purpose in the script. "**I**" is traditional because mathematicians tend to use letters between **i** and **n** to represent integer values, but in programming languages we don't have to restrict ourselves to single-letter variable names.)

---

[1] The **for** block is also in the tools library; choose "**Import tools**" from the file menu if you don't have it in the Control palette.

Global Variables

You can create variables "by hand" that aren't limited to being used within a single block. At the top of the Variables palette, click the "**Make a variable**" button:

This will bring up a dialog window in which you can give your variable a name:

The dialog also gives you a choice to make the variable available to all sprites (which is almost always what you want) or to make it visible only in the current sprite. You'd do that if you're going to give several sprites individual variables *with the same name,* so that you can share a script between sprites (by dragging it from the current sprite's scripting area to the picture of another sprite in the sprite corral), and the different sprites will do slightly different things when running that script because each has a different value for that variable name.

If you give your variable the name "**name**" then the Variables palette will look like this:

There's now a "**Delete a variable**" button, and there's an orange oval with the variable name in it, just like the orange oval in the **for** block. You can drag the variable into any script in the scripting area. Next to the oval is a checkbox, initially checked. When it's checked, you'll also see a *variable watcher* on the stage:

name

When you give the variable a value, the orange box in its watcher will display the value.

How *do* you give it a value? You use the **set** block:

ask what's-your-name? and wait
set name to answer

Note that you *don't* drag the variable's oval into the **set** block! You click on the downarrow in the first input slot, and you get a menu of all the available variable names.

Script Variables

In that example, our project is going to carry on an interaction with the user, and we want to remember her name throughout the project. That's a good example of a situation in which a *global* variable (the kind you make with the "**Make a variable**" button) is appropriate. Another common example is a variable called "**score**" in a game project. But sometimes you only need a variable temporarily, during the running of a particular script. In that case you can use the **script variables** block to make the variable:

when 🚩 clicked
script variables (wiggle) ▶
set wiggle to (pick random 1 to 10)
pen down
move 50 steps
turn ↺ (5 × wiggle) degrees
move 50 steps
turn ↻ (5 × wiggle) degrees
move 50 steps
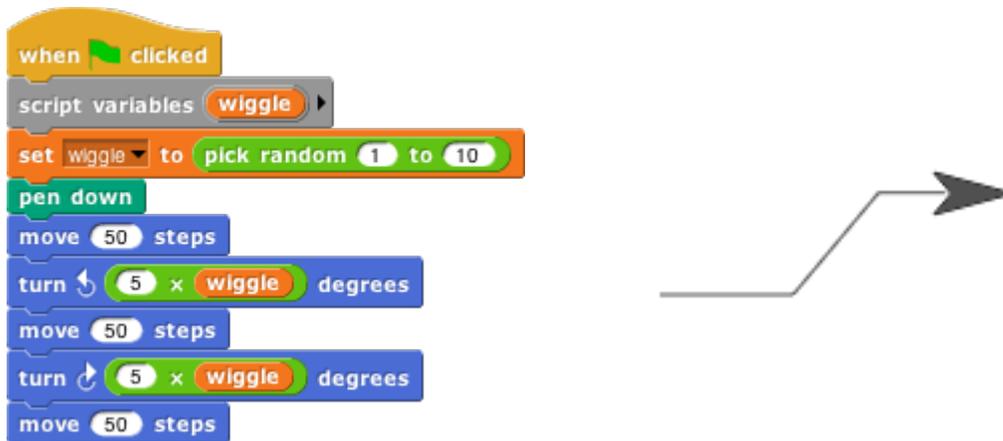
As in the **for** block, you can click on an orange oval in the **script variables** block without dragging to change its name. You can also make more than one temporary variable by clicking on the right arrow at the end of the block to add another variable oval:

script variables (a) (b) (c) ◀▶

E. Etcetera

This manual doesn't explain every block in detail. There are many more motion blocks, sound blocks, costume and graphics effects blocks, and so on. You can learn what they all do by experimentation, and also by reading the "help screens" that you can get by right-clicking or control-clicking a block and selecting "**help…**" from the menu that appears.

II. Saving and Loading Projects and Media

After you've created a project, you'll want to save it, so that you can have access to it the next time you use Snap!. There are several ways to do that. You can save a project on your own computer, or you can save it at the Snap! web site. The advantage of saving on the net is that you have access to your project even if you are using a different computer, or a mobile device such as a tablet or smartphone. The advantage of saving on your computer is that you have access to the saved project while on an airplane or otherwise not on the net. This is why we have multiple ways to save.

A. Local Storage

There are two different ways to save a project (or a media file such as a costume) on your computer. The reason for this complexity is that Javascript, in which Snap! is implemented, deliberately restricts the ability of programs running in a browser to affect the computer. This is a good thing, because it means that you can confidently run someone else's Snap! project without worrying that it will delete all your files or infect your computer with a virus. But it does make things a little complicated.

Localstore

Find the File icon (⬛) in the Tool Bar. In the menu that appears when you click it, choose the option "Save as…" You'll then see a window like this:

The "Browser" option is selected, which means that your project will be saved in a special file that can be read only on the same computer, by the same browser, connected to the same (Snap!) web page. You won't see it in a listing of your files outside of Snap!. This is how Javascript protects you against malware in saved projects.

In the picture above, the narrow space at the top is where you enter the name with which you want to save the project. Under that, on the left you see a list of projects you've already saved in the browser's storage. On the right are a picture of the stage and a text box for "project notes": any information you'd like a user of the project to know. Both of these are saved with the project.

An important limitation of this way to save projects is that your browser will set a limit on the total amount of storage available, for all web sites combined. (This limit may be settable in your browser's preferences.) So the "localstore" way to save projects (that's the official name of the browser's storage) is limited to only a few projects. Also, if your browser is configured to disallow cookies from web sites (another preference setting) then it won't allow localstore either.

XML Export

  The second way to save a project on your computer requires two steps, but it doesn't have the limitations of localstore. Projects saved in this second way are normal files on your computer and can be shared with friends, can be opened in any browser, and have no size limitation.

  From the file menu, choose "Export project…" The entire Snap! window will disappear, replaced by a screenful of what looks like gibberish. Don't panic! This is what's supposed to happen. You are looking at your project, in a notation called XML. The main reason it looks like gibberish is that it includes an encoding of the pictures and other media in the project. If you look through the XML, the actual scripts of the project are pretty readable, although they don't look like Snap! blocks. Snap! has opened a new browser tab for this XML text; the actual Snap! window is still there, hiding in its original tab.

  But the point of this XML text isn't for you to read. Once you're looking at that tab, use your browser's Save command (in its File menu, or usually with a shortcut of command-S (Mac) or control-S (everything else). You can choose a filename for it, and it'll be saved in your usual Downloads folder. You can then close that tab and return to the Snap! tab.

B. Cloud Storage

  The other possibility is to save your project "in the cloud," at the Snap! web site. In order to do this, you need an account with us. Click on the Cloud button ( ) in the Tool Bar. Choose the "Signup…" option. This will show you a window that looks like this:

You must choose a user name that will identify you on the web site, such as **Jens** or **bh**. If you're a Scratch user, you can use your Scratch name for Snap! too. If you're a kid, don't pick a user name that includes your family name, but first names or initials are okay. Don't pick something you'd be embarrassed to have other users (or your parents) see! If the name you want is already taken, you'll have to choose another one.

  We ask for your month and year of birth; we use this information only to decide whether to ask for your own email address or your parent's email address. (If you're a kid, you shouldn't sign up for anything on the net, not even Snap!, without your parent's knowledge.) We do not store your birthdate information on our server; it is used on your own computer only during this initial signup. We do not ask for your *exact* birthdate, even for this one-time purpose, because that's an important piece of personally identifiable information.

  When you click OK, an email will be sent to the email address you gave, with an initial password for your account. We keep your email address on file so that, if you forget your password, we can send you a password-reset link. We will also email you if your account is suspended for violation of the Terms of Service. We do not

14

use your address for any other purpose. You will never receive marketing emails of any kind through this site, neither from us nor from third parties. If, nevertheless, you are worried about providing this information, do a web search for "temporary email."

Finally, you must read and agree to the Terms of Service. A quick summary: Don't interfere with anyone else's use of the web site, and don't put copyrighted media or personally identifiable information in projects that you share with other users. And we're not responsible if something goes wrong.

Once you've created your account, you can log into it using the "Login…" option from the Cloud menu:

Use the user name and password that you set up earlier. If you check the "Stay signed in" box, then you will be logged in automatically the next time you run Snap! from the same browser on the same computer. Check the box if you're using your own computer and you don't share it with siblings. Don't check the box if you're using a public computer at the library, at school, etc.

Once logged in, you can choose the "Cloud" option in the "Save as…" dialog shown on page 13. You enter a project name, and optionally project notes, just as for Localstore saving, but your project will be saved online and can be loaded from anywhere with net access.

C. Loading Saved Projects

Once you've saved a project, you want to be able to load it back into Snap!. There are two ways to do this:

1. If you saved the project in Localstore or in your online Snap! account, choose the "Open…" option from the File menu. Choose the "Browser" or "Cloud" button, then select your project from the list in the big text box and click OK. (A third button, "Examples," lets you choose from example projects that we provide. You can see what each of these projects is about by clicking on it and reading its project notes.)

2. If you saved the project as an XML file on your computer, choose "Import…" from the File menu. This will give you an ordinary browser file-open window, in which you can navigate to the file as you would in other software. Alternatively, find the XML file on your desktop, and just drag it onto the Snap! window.

The second technique above also allows you to import media (costumes and sounds) into a project. Just choose "Import…" and then select a picture or sound file instead of an XML file.

Snap! can also import projects created in BYOB 3.0, or in Scratch 1.4 or (with some effort; see our web site) 2.0. Almost all such projects work correctly in Snap!, apart from a small number of incompatible blocks. BYOB 3.1 projects that don't use first class sprites work, too; all BYOB 3.1 projects will work in Snap! 4.1.

III. Building a Block

The first version of **Snap!** was called BYOB, for "Build Your Own Blocks." This was the first and is still the most important capability we added to Scratch. (The name was changed because a few teachers have no sense of humor. ☹ You pick your battles.) The new Scratch 2.0 also has a partial custom block capability.

A. Simple Blocks

In the Variables palette, at or near the bottom, is a button labeled "**Make a block**."


Clicking this button will display a dialog window in which you choose the block's name, shape, and palette/color. You also decide whether the block will be available to all sprites, or only to the current sprite and its children. Note: You can also enter the "Make a block" dialog by right-click/control-click on the script area background and then choose "**Make a block**" from the menu that appears.
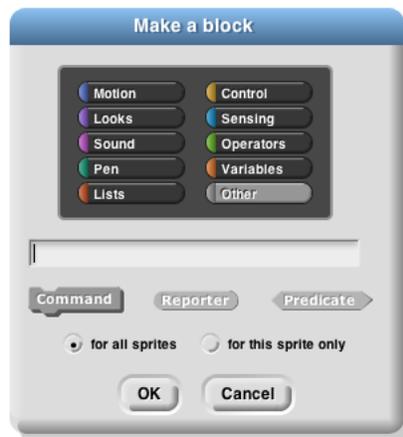

In this dialog box, you can choose the block's palette, shape and name. With one exception, there is one color per palette, e.g., all Motion blocks are blue. But the Variables palette includes the orange variable-related blocks and the red list-related blocks. Both colors are available, along with an "Other" option that makes grey blocks in the Variables palette for blocks that don't fit any category.

There are three block shapes, following a convention that should be familiar to Scratch users: The jigsaw-puzzle-piece shaped blocks are Commands, and don't report a value. The oval blocks are Reporters, and the hexagonal blocks are Predicates, which is the technical term for reporters that report Boolean (true or false) values.

Suppose you want to make a block named "square" that draws a square. You would choose Motion, Command, and type "**square**" into the name field. When you click OK, you enter the Block Editor. This works just like making a script in the sprite's scripting area, except that the "hat" block at the top, instead of saying something like "**when I am clicked**," has a picture of the block you're building. This hat block is called the *prototype* of your custom block.[1] You drag blocks under the hat to program your custom block, then click OK:
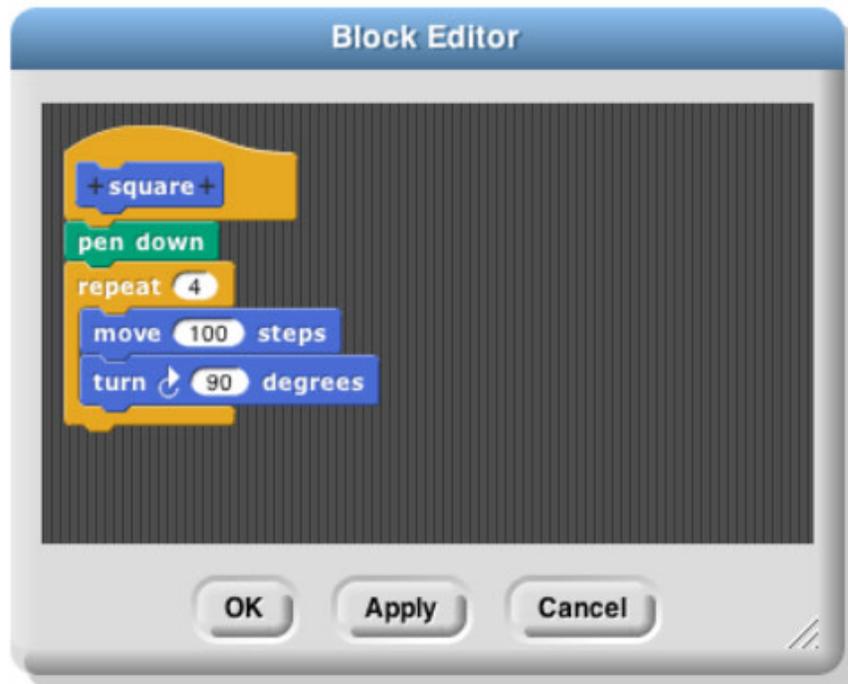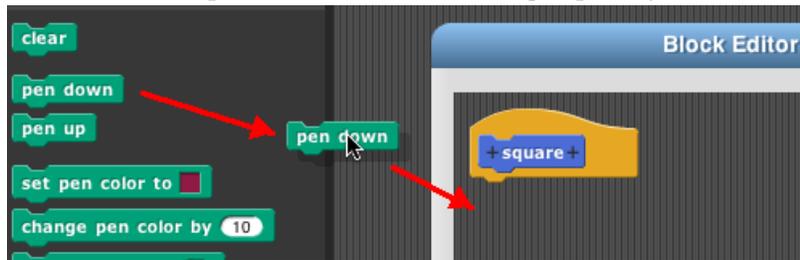

Your block appears at the bottom of the Motion palette. Here's the block and the result of using it:

---

[1] This use of the word "prototype" is unrelated to the *prototyping object oriented programming* discussed later.

Custom Blocks with Inputs

But suppose you want to be able to draw squares of different sizes. Control-click or right-click on the block, choose "**edit**," and the Block Editor will open. Notice the plus signs before and after the word **square** in the prototype block. If you hover the mouse over one, it lights up:
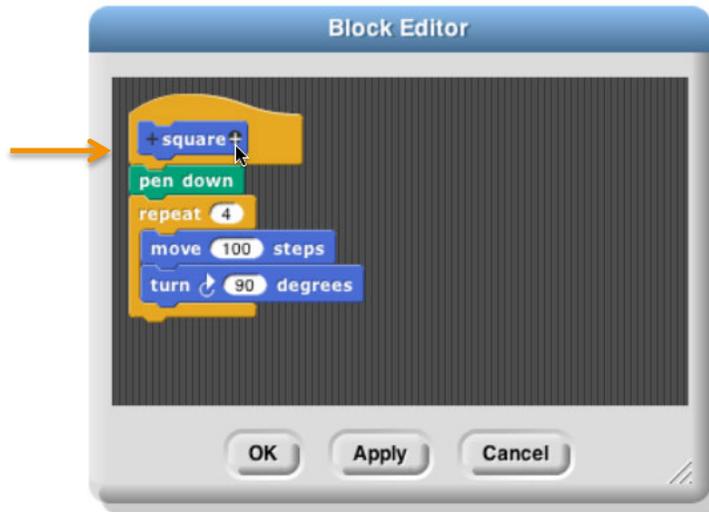
Click on the plus on the right. You will then see the "input name" dialog:
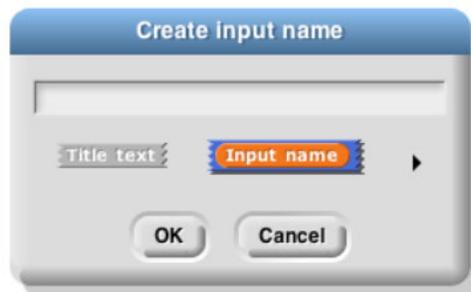
Type in the name "**size**" and click OK. There are other options in this dialog; you can choose "**title text**" if you want to add words to the block name, so it can have text after an input slot, like the "**move ( ) steps**" block. Or you can select a more extensive dialog with a lot of options about your input name. But we'll leave that for later. When you click OK, the new input appears in the block prototype:
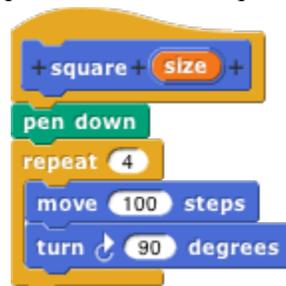
You can now drag the orange variable down into the script, then click okay:

18
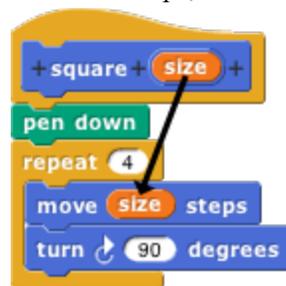
Your block now appears in the Motion palette with an input box:

You can draw any size square by entering the length of its side in the box and running the block as usual, by double-clicking it or by putting it in a script.

B. Recursion

Since the new custom block appears in its palette as soon as you *start* editing it, you can write recursive blocks (blocks that call themselves) by dragging the block into its own definition:
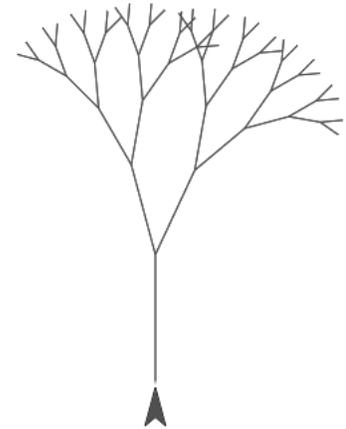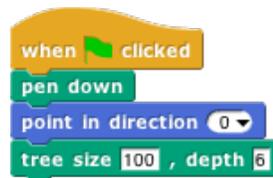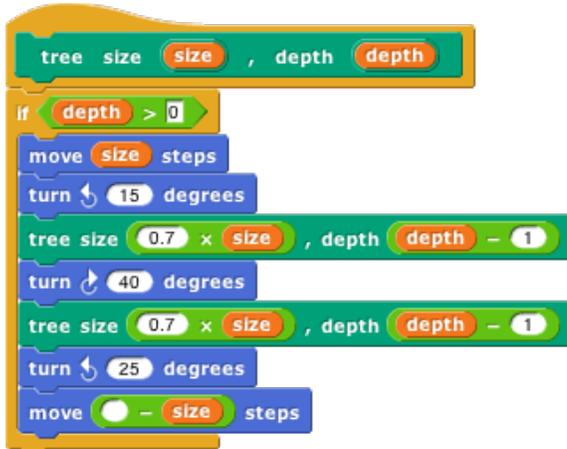
If recursion is new to you, here are a few brief hints: It's crucial that the recursion have a *base case*, that is, some small(est) case that the block can handle without using recursion. In this example, it's the case **depth**=0, for which the block does nothing at all, because of the enclosing **if**. Without a base case, the recursion would run forever, calling itself over and over.

Don't try to trace the exact sequence of steps that the computer follows in a recursive program. Instead, imagine that inside the computer there are many small people, and if Theresa is drawing a tree of size 100, depth 6, she hires Tom to make a tree of size 70, depth 5, and later hires Theo to make another tree of size 70, depth 5. Tom in turn hires Tammy and Tallulah, and so on. Each little person has his or her own local variables **size** and **depth**, each with different values.

You can also write recursive reporters, like this block to compute the factorial function:
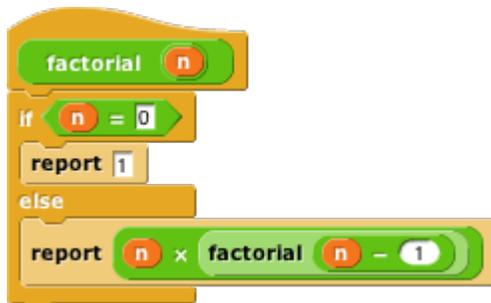
Note the use of the **report** block. When a reporter block uses this block, the reporter finishes its work and reports the value given; any further blocks in the script are not evaluated. Thus, the **if else** block in the script above could have been just an **if**, with the second **report** block below it instead of inside it, and the result would be the same, because when the first **report** is seen in the base case, that finishes the block invocation, and the second **report** is ignored. There is also a **stop block** block that has a similar purpose, ending the block

invocation early, for command blocks. (By contrast, the **stop script** block stops not only the current block invocation, but also the entire toplevel script that called it.)

Here's a slightly more compact way to write the factorial function:

(If you don't see the reporter-**if** block in the Control palette, click the file button in the tool bar and choose "**Import tools**.")

For more on recursion, see *Thinking Recursively* by Eric Roberts.

C. Block Libraries

When you save a project (see Section II above), any custom blocks you've made are saved with it. But sometimes you'd like to save a collection of blocks that you expect to be useful in more than one project. The tools library we've used throughout this manual is an example. Perhaps your blocks implement a particular data structure (a stack, or a dictionary, etc.), or they're the framework for building a multilevel game. Such a collection of blocks is called a *block library*.

Creating a block library is done using the XML Export mechanism described on page 14, except that you choose "Export blocks…" from the File menu instead of "Export project…" You then see a window like this:

The window shows all of your custom blocks. You can uncheck some of the checkboxes to select exactly which blocks you want to include in your library. Then press OK. You'll see a new tab with XML-encoded block definitions, which you save using your browser's Save command.

To import a block library, use the "Import…" command in the File menu, or just drag the XML file into the Snap! window.

IV. First Class Lists

A data type is *first class* in a programming language if data of that type can be

- the value of a variable
- an input to a procedure
- the value returned by a procedure
- a member of a data aggregate
- anonymous (not named)

In Scratch, numbers and text strings are first class. You can put a number in a variable, use one as the input to a block, call a reporter that reports a number, or put a number into a list.

But Scratch's lists are not first class. You create one using the "**Make a list**" button, which requires that you give the list a name. You can't put the list into a variable, into an input slot of a block, or into a list item—you can't have lists of lists. None of the Scratch reporters reports a list value. (You can use a reduction of the list into a text string as input to other blocks, but this loses the list structure; the input is just a text string, not a data aggregate.)

A fundamental design principle in **Snap!** is that ***all data should be first class***. If it's in the language, then we should be able to use it fully and freely. We believe that this principle avoids the need for many special-case tools, which can instead be written by **Snap!** users themselves.

Note that it's a data *type* that's first class, not an individual value. Don't think, for example, that some lists are first class, while others aren't. In **Snap!**, lists are first class, period.

A. The list Block

At the heart of providing first class lists is the ability to make an "anonymous" list—to make a list without simultaneously giving it a name. The **list** reporter block does that.
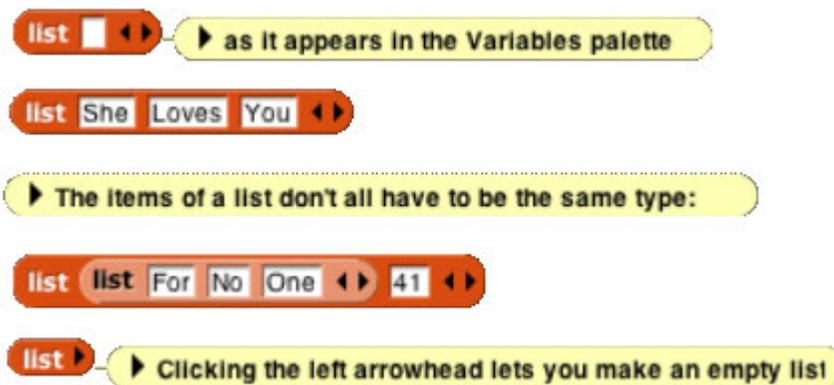
At the right end of the block are two left-and-right arrowheads. Clicking on these changes the number of inputs to **list**, i.e., the number of elements in the list you are building.

You can use this block as input to many other blocks:

Snap! does not have a "**Make a list**" button like the one in Scratch. If you want a global named list, make a global variable and use the **set** block to put a list into the variable.

B. Lists of Lists

Lists can be inserted as elements in larger lists. We can easily create ad hoc structures as needed:

We can also build any classic computer science data structure out of lists of lists, by defining *constructors* (blocks to make an example of the structure), *selectors* (blocks to pull our a piece of the structure), and *mutators* (blocks to change the contents of the structure) as needed. Here we create binary trees with selectors that check for input of the correct data type; only one selector is shown but the ones for left and right children are analogous.
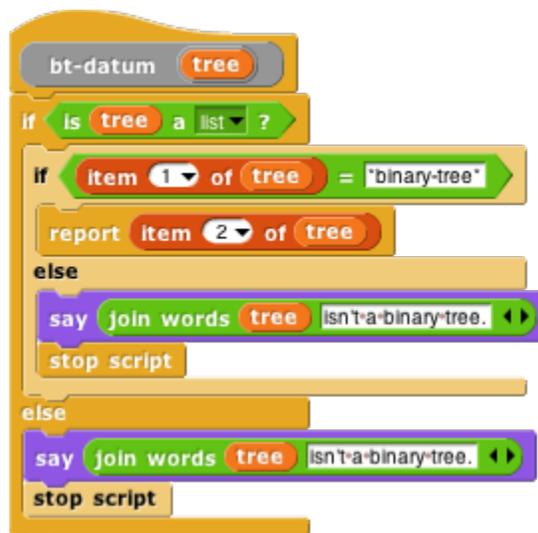
C. Functional and Imperative List Programming

There are two ways to create a list inside a program. Scratch users will be familiar with the *imperative* programming style, which is based on a set of command blocks that modify a list:
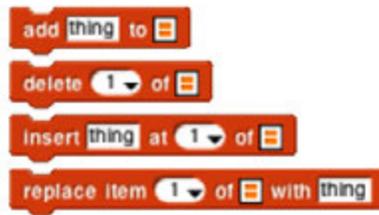
As an example, here is a block that takes a list of numbers as input, and reports a new list containing only the even numbers from the original list:
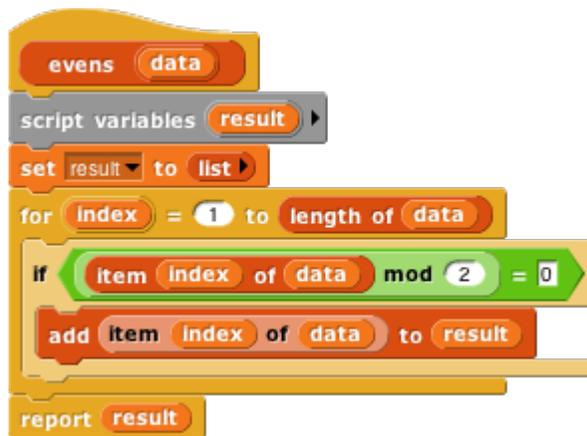
In this script, we first create a temporary variable, then put an empty list in it, then go through the items of the input list using the **add … to (result)** block to modify the result list, adding one item at a time, and finally report the result.

Functional programming is a different approach that is becoming important in "real world" programming because of parallelism, i.e., the fact that different processors can be manipulating the same data at the same time. This makes the use of mutation (changing the value associated with a variable) problematic because it's impossible to know the exact sequence of events, so the result of mutation may not be what the programmer expected. Even without parallelism, though, functional programming is sometimes a simpler and more effective technique, especially when dealing with recursively defined data structures. It uses reporter blocks, not command blocks, to build up a list value:
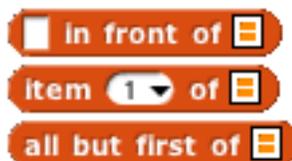
In a functional program, we often use recursion to construct a list, one item at a time. The **in front of** block makes a list that has one item added to the front of an existing list, *without changing the value of the original list.* A nonempty list is processed by dividing it into its first item (**item 1 of**) and all the rest of the items (**all but first of**), which are handled through a recursive call:
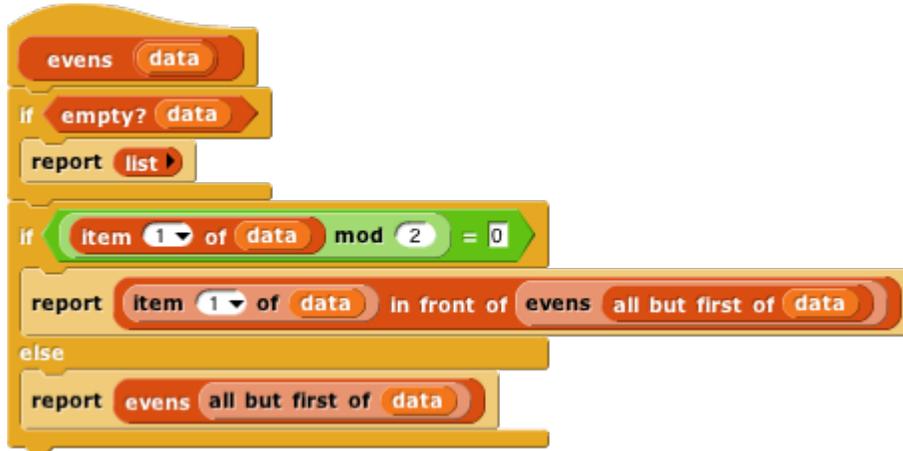
Snap! uses two different internal representations of lists, one (dynamic array) for imperative programming and the other (linked list) for functional programming. Each representation makes the corresponding built-in list blocks (commands or reporters, respectively) most efficient. It's possible to mix styles in the same program, but if the same list is used both ways, the program will run more slowly because it converts from one representation to the other repeatedly. (The **item ( ) of [ ]** block doesn't change the representation.) You don't have to know the details of the internal representations, but it's worthwhile to use each list in a consistent way.

D. Higher Order List Operations and Rings

There's an even easier way to select the even numbers from a list:

(If you don't have the **keep** block near the bottom of the Variable palette, click the File button in the tool bar and select "**Import tools…**")

The **keep** block takes a Predicate expression as its first input, and a list as its second input. It reports a list containing those elements of the input list for which the predicate returns **true**. Notice two things about the predicate input: First, it has a grey ring around it. Second, the **mod** block has an empty input. **Keep** puts each item of its input list, one at a time, into that empty input before evaluating the predicate. (The empty input is supposed to remind you of the "box" notation for variables in elementary school: □+3=7.) The grey ring is part of the **keep** block as it appears in the palette:

What the ring means is that this input is a block (a predicate block, in this case, because the interior of the ring is a hexagon), rather than the value reported by that block. Here's the difference:

Evaluating the **=** block without a ring reports a number; evaluating the block *with* a ring reports the block itself. This allows **keep** to evaluate the **=** predicate repeatedly, once for each list item. A block that takes another block as input is called a *higher order* block.

Snap! provides three higher order blocks for operating on lists:

map ( ) over [list] ◀▶

keep items such that ( ) from [list]

combine with ( ) items of [list]
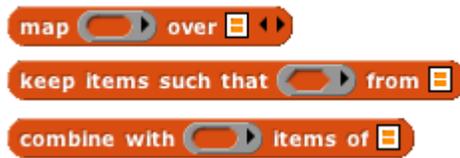
You've already seen **keep**. **Map** takes a Reporter block and a list as inputs. It reports a new list in which each item is the value reported by the Reporter block as applied to one item from the input list. That's a mouthful, but an example will make its meaning clear:

map (( ) × 10) over (list 7 8 1 ◀▶) ◀▶

1	70	-
2	80	-
3	10	-
⊕	length: 3	

By the way, we've been using arithmetic examples, but the list items can be of any type, and any reporter can be used. We'll make the plurals of some words:

map (join [ ] s ◀▶) over (list book computer bagel ◀▶) ◀▶

1	books	-
2	computers	
3	bagels	-
⊕	length: 3	

These examples use small lists, to fit the pages, but the higher order blocks work for any size list.

The **map** block has arrowheads at the right end because of a little-used feature that allows mapping through multiple same-length lists in parallel, with a multi-input reporter expression. Don't worry about it for now; just make sure when you drag an expression into the list input slot that you hit the slot and not the arrowheads.

The third higher order block, **combine**, computes a single result from *all* the items of a list, using a *two-input* reporter as its first input. In practice, there are only a few blocks you'll ever use with **combine**:

( ) + ( )   ( ) × ( )   join [ ] [ ] ◀▶   join words [ ] [ ] ◀▶   ⬗ and ⬗   ⬗ or ⬗

These blocks take the sum of the list items, take their product, string them into one word, combine them into a sentence (with spaces between items), see if all items of a list of Booleans are true, or see if any of the items is true.

combine with (( ) + ( )) items of (list 7 8 1 ◀▶)   `16`

Why **+** but not **-**? It only makes sense to combine list items using an *associative* function: one that doesn't care in what order the items are combined (left to right or right to left). (2+3)+4 = 2+(3+4), but (2−3)−4 ≠ 2−(3−4).

V. Typed Inputs

A. Scratch's Type Notation

Scratch block inputs come in two types: Text-or-number type and Number type. The former is indicated by a rectangular box, the latter by a rounded box: letter 1 of world . A third Scratch type, Boolean (true/false), can be used in certain Control blocks with hexagonal slots.

The Snap! types are an expanded collection including Procedure, List, and Object types. Note that, with the exception of Procedure types, all of the input type shapes are just reminders to the user of what the block expects; they are not enforced by the language.

B. The Snap! Input Type Dialog

In the Block Editor input name dialog, there is a right-facing arrowhead after the "Input name" option:

Clicking that arrowhead opens the "long" input name dialog:
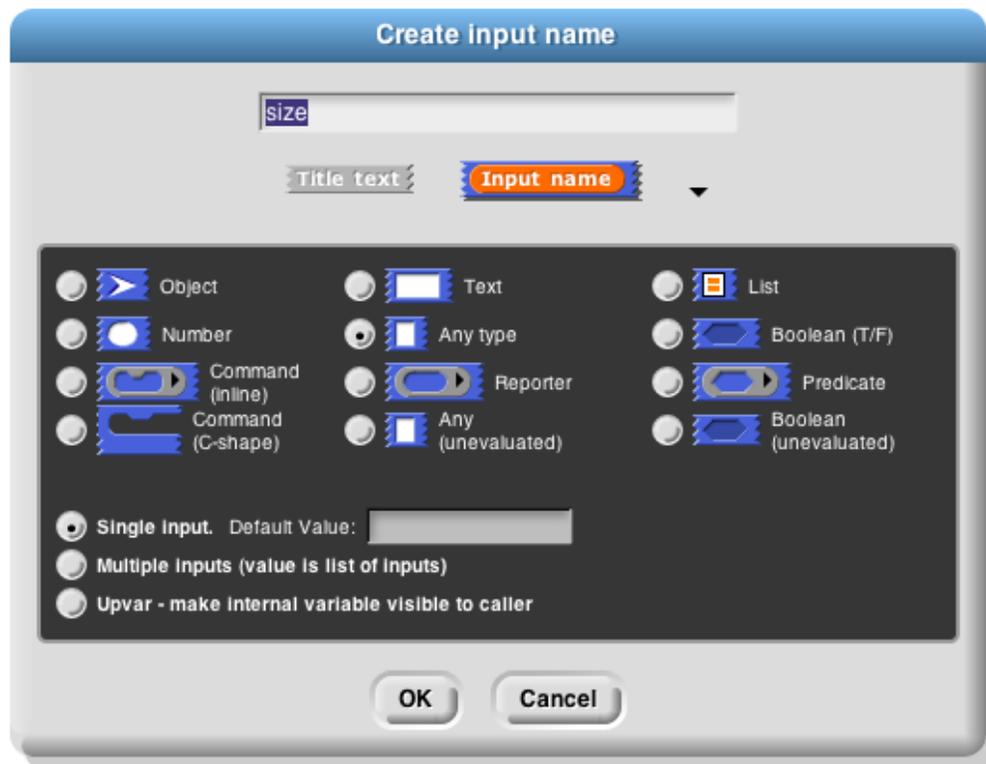
There are twelve input type shapes, plus three mutually exclusive categories, listed in addition to the basic choice between title text and an input name. The default type, the one you get if you don't choose anything else, is "Any," meaning that this input slot is meant to accept any value of any type. If the size input in your block should be an oval-shaped numeric slot rather than a generic rectangle, click "Number."

The arrangement of the input types is systematic. As the pictures on this and the next page show, each row of types is a category, and parts of each column form a category. Understanding the arrangement will make it a little easier to find the type you want.
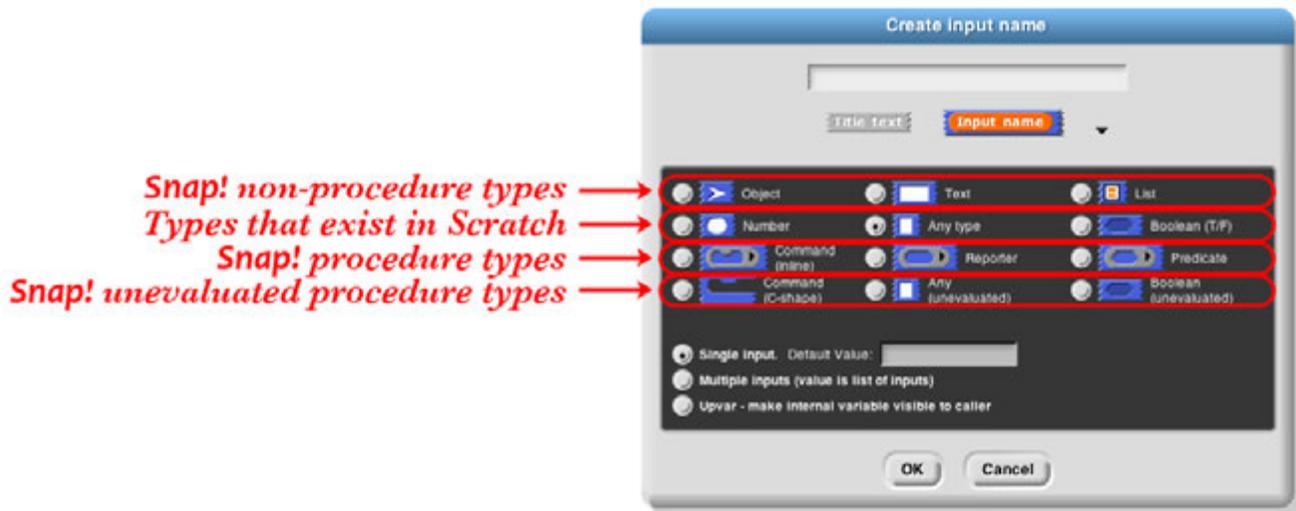
The second row of input types contains the ones found in Scratch: Number, Any, and Boolean. (The reason these are in the second row rather than the first will become clear when we look at the column arrangement.) The first row contains the new **Snap!** types other than procedures: Object, Text, and List. The last two rows are the types related to procedures, discussed more fully in section VI below.

The List type is used for first class lists, discussed in section IV above. The red rectangles inside the input slot are meant to resemble the appearance of lists as **Snap!** displays them on the stage: each element in a red rectangle.

The Object type is reserved for first class sprites, coming in **Snap!** 4.1.

The Text type is really just a variant form of the Any type, using a shape that suggests a text input.[1]

Procedure Types

Although the procedure types are discussed more fully later, they are the key to understanding the column arrangement in the input types. Like Scratch, **Snap!** has three block shapes: jigsaw-piece for command blocks, oval for reporters, and hexagonal for predicates. (A *predicate* is a reporter that always reports **true** or **false**.) In **Snap!** these blocks are first class data; an input to a block can be of Command type, Reporter type, or Predicate type. Each of these types is directly below the type of value that that kind of block reports, except for Commands, which don't report a value at all. Thus, oval Reporters are related to the Any type, while hexagonal Predicates are related to the Boolean (true or false) type.

---

[1] In Scratch, every block that takes a Text-type input has a default value that makes the rectangles for text wider than tall. The blocks that aren't specifically about text either are of Number type or have no default value, so those rectangles are taller than wide. At first we thought that Text was a separate type that always had a wide input slot; it turns out that this isn't true in Scratch (delete the default text and the rectangle narrows), but we thought it a good idea anyway, so we allow Text-shaped boxes even for empty input slots. (This is why Text comes just above Any in the input type selection box.)

The unevaluated procedure types in the fourth row are explained in section VI.E below. In one handwavy sentence, they combine the *meaning* of the procedure types with the *appearance* of the reported value types two rows higher. (Of course, this isn't quite right for the C-shaped command input type, since commands don't report values. But you'll see later that it's true in spirit.)

Input Variants

We now turn to the three mutually exclusive options that come below the type array.

The "**Single input**" option: In Scratch all inputs are in this category. There is one input slot in the block as it appears in its palette. If a single input is of type Any, Number, or Text, then you can specify a default value that will be shown in that slot in the palette, like the "10" in the **move (10) steps** block. In the prototype block at the top of the script in the Block editor, an input with name "size" and default value 10 looks like this:

The "**Multiple inputs**" option: The **list** block introduced earlier accepts any number of inputs to specify the items of the new list. To allow this, **Snap!** introduces the arrowhead notation (◀ ▶) that expands and contracts the block, adding and removing input slots. Custom blocks made by the **Snap!** user have that capability, too. If you choose the "**Multiple inputs**" button, then arrowheads will appear after the input slot in the block. More or fewer slots (as few as zero) may be used. When the block runs, all of the values in all of the slots for this input name are collected into a list, and the value of the input as seen inside the script is that list of values:

The ellipsis (…) in the orange input slot name box in the prototype indicates a Multiple input.

The third category, "**Upvar - make internal variable visible to caller**," isn't really an input at all, but rather a sort of output from the block to its user. It appears as an orange variable oval in the block, rather than as an input slot. Here's an example; the uparrow (↑) in the prototype indicates this kind of internal variable name:

The variable **i** (in the block on the right above) can be dragged from the **for** block into the blocks used in its C-shaped command slot. Also, by clicking on the orange **i**, the user can change the name of the variable as seen in the calling script (although the name hasn't changed inside the block's definition). This kind of variable is called an *upvar* for short, because it is passed *upward* from the custom block to the script that uses it.

Prototype Hints

We have noted three notations that can appear in an input slot in the prototype to remind you of what kind of input this is. Here is the complete list of such notations:

=	default value	…	multiple inputs	↑	upvar
λ	procedure types	⋮	list	?	Boolean

VI. Procedures as Data

A. Call and Run

In the **for** block example above, the input named **action** has been declared as type "Command (C-shaped)"; that's why the finished block is C-shaped. But how does the block actually tell **Snap!** to carry out the commands inside the C-slot? Here is a simple version of the block script:
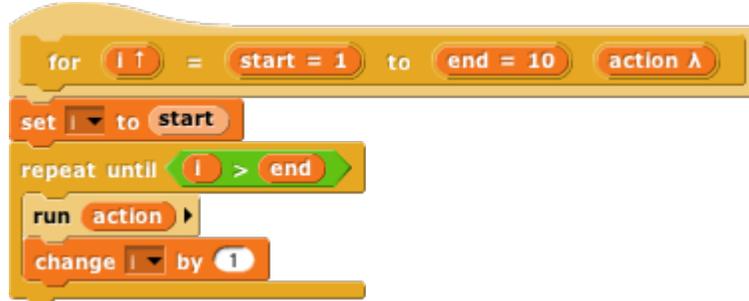
This is simplified because it assumes, without checking, that the ending value is greater than the starting value; if not, the block should (depending on the designer's purposes) either not run at all, or change the variable by −1 instead of by 1. (The **for** block in **Snap!**'s tool library works for ascending or descending values; you can read its script by right-clicking or control-clicking on it and selecting the Edit option.)

The important part of this script is the **run** block near the end. This is a **Snap!** built-in command block that takes a Command-type value (a script) as its input, and carries out its instructions. (In this example, the value of the input **action** is the script that the user puts in the C-slot of the **for** block.) There is a similar **call** reporter block for invoking a Reporter or Predicate block. The **call** and **run** blocks are at the heart of **Snap!**'s first class procedure feature; they allow scripts and blocks to be used as data — in this example, as an input to a block — and eventually carried out under control of the user's program.

Here's another example, this time using a Reporter-type input in a simplified **map** block (see page 22):
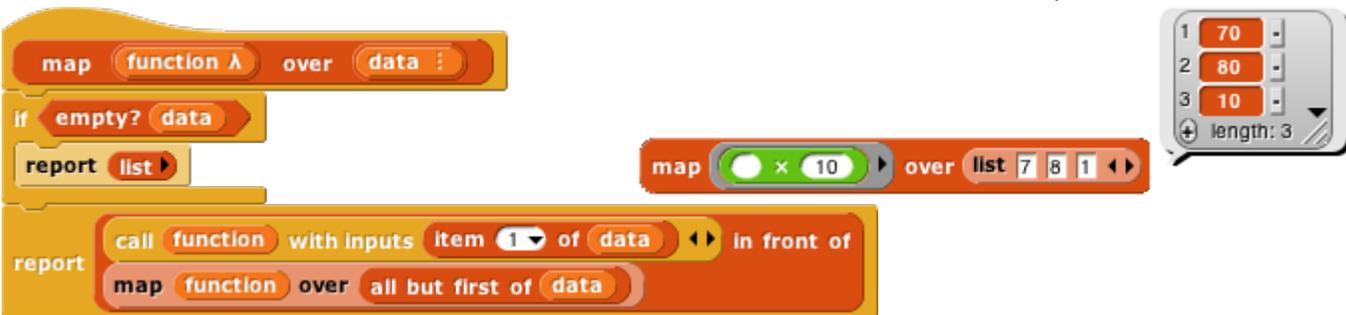
Here we are calling the Reporter "multiply by 10" three times, once with each item of the given list as its input, and collecting the results as a list. (The reported list will always be the same length as the input list.) Note that the multiplication block has two inputs, but here we have specified a particular value for one of them (10), so the **call** block knows to use the input value given to it just to fill the other (empty) input slot in the multiplication block. In the **map** definition, the input **function** is declared to be type Reporter, and **data** is of type List.

Call/Run with inputs

The **call** block (like the **run** block) has a right arrowhead at the end; clicking on it adds the phrase "with inputs" and then a slot into which an input can be inserted:

If the left arrowhead is used to remove the last input slot, the "with inputs" disappears also. The right arrowhead can be clicked as many times as needed for the number of inputs required by the reporter block being called.

If the number of inputs given to **call** (not counting the Reporter-type input that comes first) is the same as the number of empty input slots, then the empty slots are filled from left to right with the given input values. If **call** is given exactly one input, then *every* empty input slot of the called block is filled with the same value:

An even more important thing to notice about these examples is the *ring* around the Reporter-type input slots in **call** and **map** above. This notation indicates that *the block itself,* not the number or other value that the block would report when called, is the input. If you want to use a block itself in a non-Reporter-type (e.g., Any-type) input slot, you can enclose it explicitly in a ring, found at the top of the Operators palette.
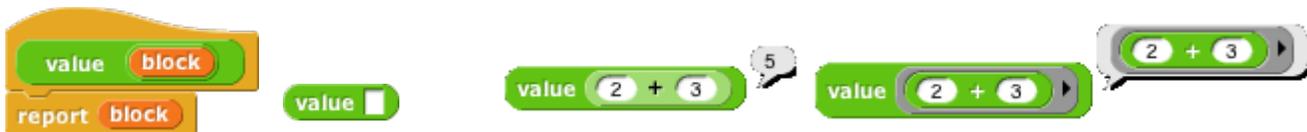
As a shortcut, if you right-click or control-click on a block (such as the **+** block in this example), one of the choices in the menu that appears is "**ringify**" or "**unringify**." The ring indicating a Reporter-type or Predicate-type input slot is essentially the same idea for reporters as the C-shaped input slot with which you're already familiar; with a C-shaped slot, it's *the script* you put in the slot that becomes the input to the C-shaped block.

Variables in Ring Slots

Note that the **run** block in the definition of the **for** block (page 30) doesn't have a ring around its input variable **action**. When you drag a variable into a ringed input slot, you generally *do* want to use *the value of* the variable, which will be the block or script you're trying to run or call, rather than the orange variable reporter itself. So Snap! automatically removes the ring in this case. If you ever do want to use the variable *block itself,* rather than the value of the variable, as a Procedure-type input, you can drag the variable into the input slot, then control-click or right-click it and choose "**ringify**" from the menu that appears. (Similarly, if you ever want to call a function that will report a block to use as the input, you can choose "**unringify**" from the menu. Almost all the time, though, Snap! does what you mean without help.)

B. Writing Higher Order Procedures

A *higher order procedure* is one that takes another procedure as an input, or that reports a procedure. In this document, the word "procedure" encompasses scripts, individual blocks, and nested reporters. (Unless specified otherwise, "reporter" includes predicates. When the word is capitalized inside a sentence, it means specifically oval-shaped blocks. So, "nested reporters" includes predicates, but "a Reporter-type input" doesn't.)

Although an Any-type input slot (what you get if you use the small input-name dialog box) will accept a procedure input, it doesn't automatically ring the input as described above. So the declaration of Procedure-type inputs makes the use of your custom higher order block much more convenient.

Why would you want a block to take a procedure as input? This is actually not an obscure thing to do; the Scratch conditional and looping blocks (the C-shaped ones in the Control palette) take a script as input. Scratch users just don't usually think about it in those terms! We could write the **repeat** block as a custom block this way, if **Snap!** didn't already have one:
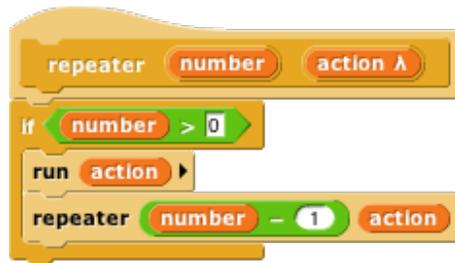
The lambda (λ) next to **action** in the prototype indicates that this is a C-shaped block, and that the script enclosed by the C when the block is used is the input named **action** in the body of the script. The only way to make sense of the variable **action** is to understand that its value is a script.

To declare an input to be Procedure-type, open the input name dialog as usual, and click on the arrowhead:

Then, in the long dialog, choose the appropriate Procedure type. The third row of input types has a ring in the shape of each block type (jigsaw for Commands, oval for Reporters, and hexagonal for Predicates). In practice, though, in the case of Commands it's more common to choose the C-shaped slot on the fourth row, because this "container" for command scripts is familiar to Scratch users. Technically the C-shaped slot is an *unevaluated* procedure type, something discussed in section E below. The two Command-related input types (inline and C-shaped) are connected by the fact that if a variable, an **item (#) of [list]** block, or a custom Reporter block is dropped onto a C-shaped slot, it turns into an inline slot, as in the **repeater** block's recursive call above. (Other built-in Reporters can't report scripts, so they aren't accepted in a C-shaped slot.)
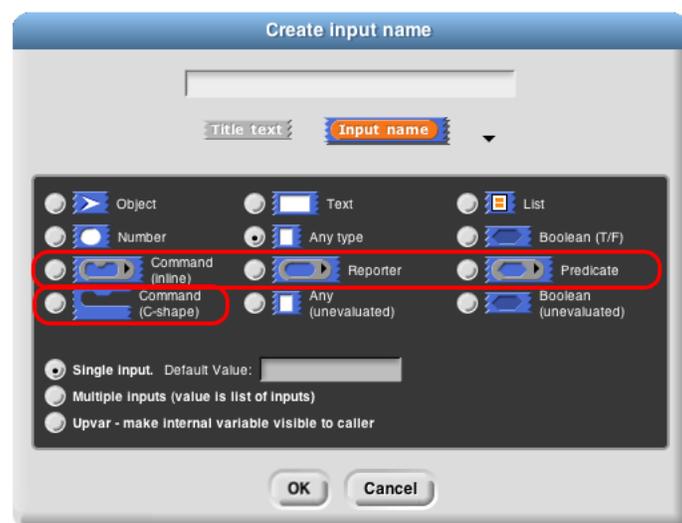
Why would you ever choose an inline Command slot rather than a C shape? Other than the **run** block discussed below, the only case I can think of is something like the C/C++/Java **for** loop, which actually has three command script inputs (and one predicate input), only one of which is the "featured" loop body:

Okay, now that we have procedures as inputs to our blocks, how do we use them? We use the blocks **run** (for commands) and **call** (for reporters). The **run** block's script input is an inline ring, not C-shaped, because we anticipate that it will be rare to use a specific, literal script as the input. Instead, the input will generally be a variable whose *value* is a script.

The **run** and **call** blocks have arrowheads at the end that can be used to open slots for inputs to the called procedures. How does **Snap!** know where to use those inputs? If the called procedure (block or script) has empty input slots, **Snap!** "does the right thing." This has several possible meanings:

1. If the number of empty slots is exactly equal to the number of inputs provided, then **Snap!** fills the empty slots from left to right:

2. If exactly one input is provided, **Snap!** will fill any number of empty slots with it:

3. Otherwise, **Snap!** won't fill any slots, because the user's intention is unclear.

If the user wants to override these rules, the solution is to use a ring with explicit input names that can be put into the given block or script to indicate how inputs are to be used. This will be discussed more fully below.

Recursive Calls to Multiple-Input Blocks

A relatively rare situation not yet considered here is the case of a recursive block that has a variable number of inputs. Let's say the user of your project calls your block with five inputs one time, and 87 inputs another time. How do you write the recursive call to your block when you don't know how many inputs to give it? The answer is that you collect the inputs in a list (recall that, when you declare an input name to represent a variable number of inputs, your block sees those inputs as a list of values in the first place), and then, in the recursive call, you drop that input list *onto the arrowheads* that indicate a variable-input slot, rather than onto the input slot:
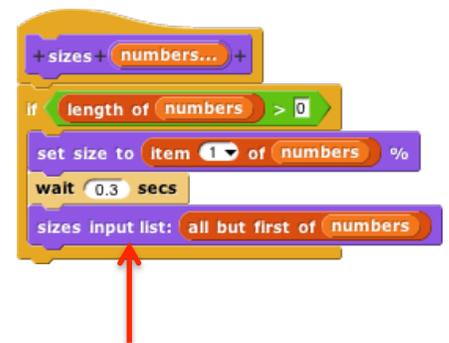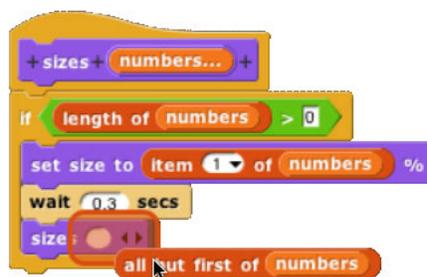
Note that the halo you see while dragging onto the arrowheads is red instead of white, and covers the input slot as well as the arrowheads. And when you drop the expression onto the arrowheads, the words "**input list:**" are added to the block text and the arrowheads disappear (in this invocation only) to remind you that the list represents all of the multiple inputs, not just a single input. The items in the list are taken *individually* as inputs to the script. Since **numbers** is a list of numbers, each individual item is a number, just what **sizes** wants. This block will take any number of numbers as inputs, and will make the sprite grow and shrink accordingly:

The user of this block calls it with any number of *individual numbers* as inputs. But inside the definition of the block, all of those numbers form *a list* that has a single input name, **numbers**. This recursive definition first checks to make sure there are any inputs at all. If so, it processes the first input (**item 1 of** the list), then it wants to make a recursive call with all but the first number. But **sizes** doesn't take a list as input; it takes numbers as inputs! So this would be wrong:

C. Formal Parameters

The rings around Procedure-type inputs have an arrowhead at the right. Clicking the arrowhead allows you to give the inputs to a block or script explicit names, instead of using empty input slots as we've done until now.

The names **#1**, **#2**, etc. are provided by default, but you can change a name by clicking on its orange oval in the **input names** list. Be careful not to *drag* the oval when clicking; that's how you use the input inside the ring. The names of the input variables are called the *formal parameters* of the encapsulated procedure.

Here's a simple but contrived example using explicit names to control which input goes where inside the ring:

Here we just want to put one of the inputs into two different slots. If we left all three slots empty, Snap! would not fill any of them, because the number of inputs provided (2) would not match the number of empty slots (3).

Here is a more realistic, much more advanced example:
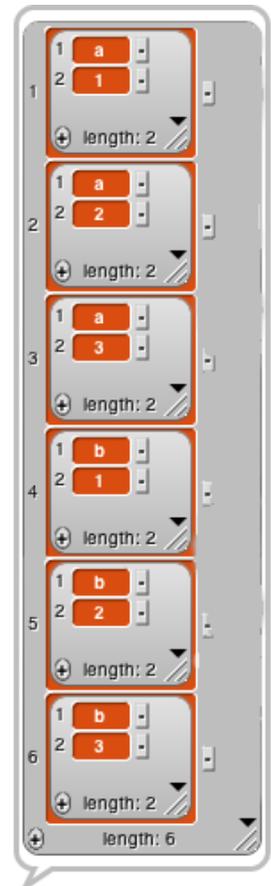
This is the definition of a block that takes any number of lists, and reports the list of all possible combinations of one item from each list. The important part for this discussion is that near the bottom there are two *nested* calls to **map**, the higher order function that applies an input function to each item of an input list. In the inner block, the function being mapped is **in front of**, and that block takes two inputs. The second, the empty List-type slot, will get its value in each call from an item of the inner **map**'s list input. But there is no way for the outer **map** to communicate values to empty slots of the **in front of** block. We must give an explicit name, **newitem**, to the value that the outer **map** is giving to the inner one, then drag that variable into the **in front of** block.

By the way, once the called block provides names for its inputs, Snap! will not automatically fill empty slots, on the theory that the user has taken control. In fact, that's another reason you might want to name the inputs explicitly: to stop Snap! from filling a slot that should really remain empty.

D. Procedures as Data

Here's an example of a situation in which a procedure must be explicitly marked as data by pulling a ring from the Operators palette and putting the procedure (block or script) inside it:
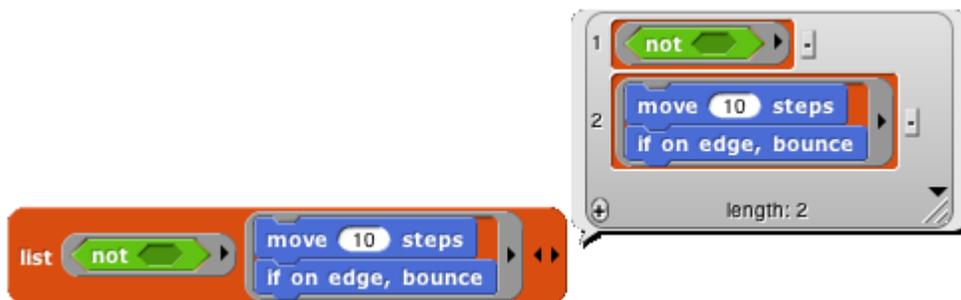
Here, we are making a list of procedures. But the **list** block accepts inputs of any type, so its input slots are not ringed. We must say explicitly that we want the block *itself* as the input, rather than whatever value would result from evaluating the block.

Besides the **list** block in the example above, other blocks into which you may want to put procedures are **set** (to set the value of a variable to a procedure), **say** and **think** (to display a procedure to the user), and **report** (for a reporter that reports a procedure):

E. Special Forms

Scratch has an **if else** block that has two C-shaped command slots and chooses one or the other depending on a Boolean test. Because Scratch doesn't emphasize functional programming, it lacks a corresponding reporter block to choose between two expressions. We could write one in Snap!:
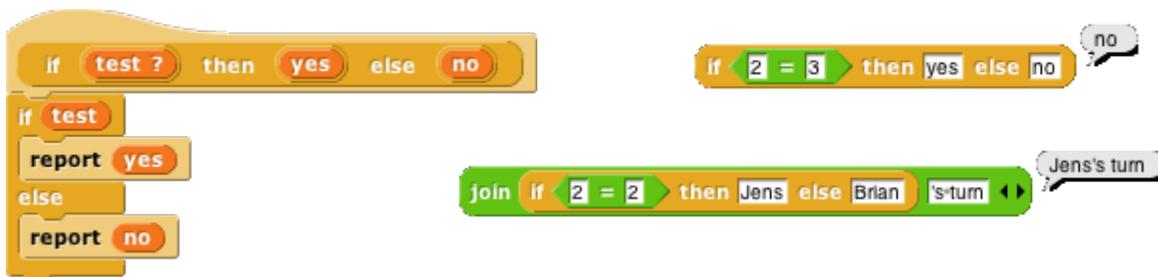
Our block works for these simple examples, but if we try to use it in writing a recursive operator, it'll fail:
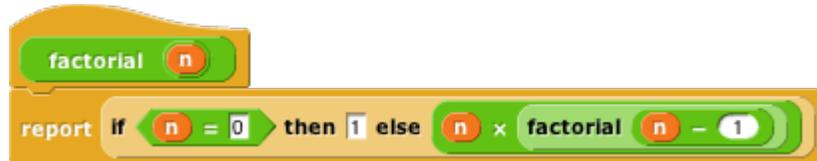
The problem is that when any block is called, all of its inputs are computed (evaluated) before the block itself runs. The block itself only knows the values of its inputs, not what expressions were used to compute them. In particular, all of the inputs to our **if then else** block are evaluated first thing. That means that even in the base case, **factorial** will try to call itself recursively, causing an infinite loop. We need our **if then else** block to be able to select only one of the two alternatives to be evaluated.

We have a mechanism to allow that: declare the **then** and **else** inputs to be of type Reporter rather than type Any. Then, when calling the block, those inputs will be enclosed in a ring so that the expressions themselves, rather than their values, become the inputs:

36
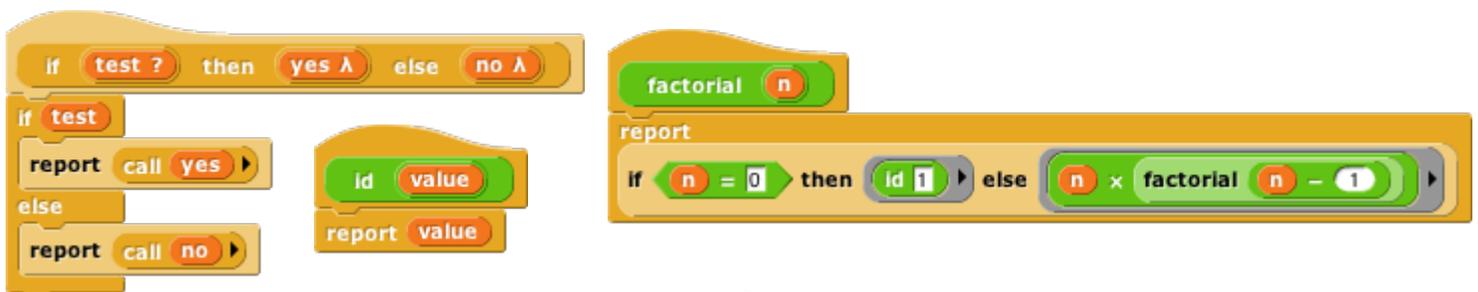
In this version, the program works, with no infinite loop. But we've paid a heavy price: this reporter-**if** is no longer as intuitively obvious as the Scratch command-**if**. You have to know about procedures as data, about rings, and about a trick to get a constant value in a ringed slot. (The **id** block implements the identity function, which reports its input. We need it because rings only take reporters as input, not numbers.) What we'd like is a reporter-**if** that *behaves* like this one, delaying the evaluation of its inputs, but *looks* like our first version, which was easy to use except that it didn't work.

Such blocks are indeed possible. A block that seems to take a simple expression as input, but delays the evaluation of that input by wrapping an "invisible ring" around it (and, if necessary, an **id**-like transformation of constant data into constant functions) is called a *special form*. To turn our **if** block into a special form, we edit the block's prototype, declaring the inputs **yes** and **no** to be of type "**Any (unevaluated)**" instead of type Reporter. The script for the block is still that of the second version, including the use of **call** to evaluate either **yes** or **no** but not both. But the slots appear as white Any-type rectangles, not Reporter-type rings, and the **factorial** block will look like our first attempt.

In a special form's prototype, the unevaluated input slot(s) are indicated by a lambda (λ) next to the input name, just as if they were declared as Procedure type. They *are* Procedure type, really; they're just disguised to the user of the block.

Special forms trade off implementor sophistication for user sophistication. That is, you have to understand all about procedures as data to make sense of the special form implementation of **if then else**. But any experienced Scratch programmer can *use* **if then else** without thinking at all about how it works internally.

Special Forms in Scratch

Special forms are actually not a new invention in **Snap!**. Many of Scratch's conditional and looping blocks are really special forms. The hexagonal input slot in the **if** block is a straightforward Boolean value, because the value can be computed once, before the **if** block makes its decision about whether or not to run its action input. But the **forever if**, **repeat until**, and **wait until** blocks' inputs can't be Booleans; they have to be of type "Boolean (unevaluated)," so that Scratch can evaluate them over and over again. Since Scratch doesn't have custom blocks, it can afford to handwave away the distinction between evaluated and unevaluated Booleans, but **Snap!** can't. The pedagogic value of special forms is proven by the fact that no Scratcher ever notices that there's anything strange about the way in which the hexagonal inputs in the Control blocks are evaluated.

Also, the C-shaped slot familiar to Scratch users is an unevaluated procedure type; you don't have to use a ring to keep the commands in the C-slot from being run before the C-shaped block is run. Those commands themselves, not the result of running them, are the input to the C-shaped Control block. (This is taken for granted by Scratch users, especially because Commands don't report values, so it wouldn't make sense to think of putting commands in the C-shaped slot as a composition of functions.) This is why it makes sense that "C-shaped" is on the fourth row of types in the long form input dialog, with other unevaluated types.

VII. Object Oriented Programming

   Object oriented programming is a style based around the abstraction *object:* a collection of *data* and *methods* (procedures, which from our point of view are just more data) that you interact with by sending it a *message* (just a name, maybe in the form of a text string, and perhaps additional inputs). The object responds to the message by carrying out a method, which may or may not report a value back to the asker. Some people emphasize the *data hiding* aspect of OOP (because each object has local variables that other objects can access only by sending request messages to the owning object) while others emphasize the *simulation* aspect (in which each object abstractly represents something in the world, and the interactions of objects in the program model real interactions of real people or things). Data hiding is important for large multi-programmer industrial projects, but for **Snap!** users it's the simulation aspect that's important. Our approach is therefore less restrictive than that of some other OOP languages.

   Technically, object oriented programming rests on three legs: (1) *Message passing:* There is a notation by which any object can send a message to another object. (2) *Local state:* Each object can remember the important past history of the computation it has performed. ("Important" means that it need not remember every message it has handled, but only the lasting effects of those messages that will affect later computation.) (3) *Inheritance:* It would be impractical if each individual object had to contain methods, many of them identical to those of other objects, for all of the messages it can accept. Instead, we need a way to say that this new object is just like that old object except for a few differences, so that only those differences need be programmed explicitly.

   The idea of object oriented programming is often taught in a way that makes it seem as if a special object oriented programming language is necessary. In fact, any language with first class procedures allows objects to be implemented explicitly; this is a useful exercise to help demystify objects.

   The central idea of this implementation is that an object is represented as a *dispatch procedure* that takes a message as input and reports the corresponding method. In this section we start with a stripped-down example to show how local state works, and build up to full implementations of class/instance and prototyping OOP.

A. Local State with Script Variables
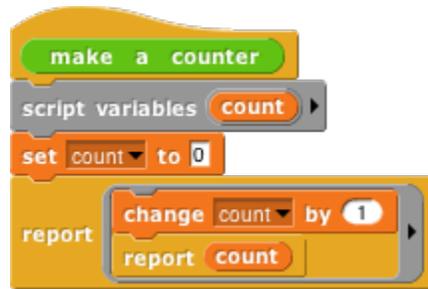
This script implements an object *class*, a type of object, namely the counter class. In this first simplified version there is only one method, so no explicit message passing is necessary. When the **make a counter** block is called, it reports a procedure, the ringed script inside its body. That procedure implements a specific counter object, an *instance* of the counter class. When invoked, a counter instance increases and reports its count variable. Each counter has its own local count:

This example will repay careful study, because it isn't obvious why each instance has a separate count. From the point of view of the **make a counter** procedure, each invocation causes a new **count** variable to be created. Usually such *script variables* are temporary, going out of existence when the script ends. But this one is special, because **make a counter** returns another script that makes reference to the **count** variable, so it remains active. (The **script variables** block makes variables local to a script. It can be used in a sprite's script area or in the Block Editor. Script variables can be "exported" by being used in a reported procedure, as here.)

In this approach to OOP, we are representing both classes and instances as procedures. The **make a counter** block represents the class, while each instance is represented by a nameless script created each time **make a counter** is called. The script variables created inside the **make a counter** block but outside the ring are *instance variables,* belonging to a particular counter.

B. Messages and Dispatch Procedures

In the simplified class above, there is only one method, and so there are no messages; you just call the instance to carry out its one method. Here is a more refined version that uses message passing:
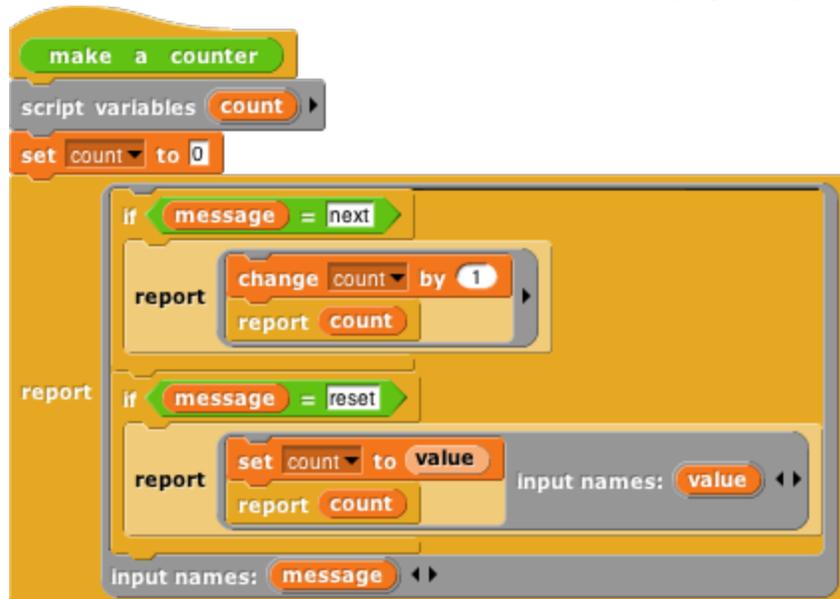
Again, the **make a counter** block represents the **counter** class, and again the script creates a local variable **count** each time it is invoked. The large outer ring represents an instance. It is a *dispatch procedure:* it takes a **message** (just a text word) as input, and it reports a method. The two smaller rings are the methods. The top one is the **next** method; the bottom one is the **reset** method. The latter requires an input, named **value**.

In the earlier version, calling the instance did the entire job. In this version, calling the instance gives access to a method, which must then be called to finish the job. We can provide a block to do both procedure calls in one:

The **ask** block has two required inputs: an object and a message. It also accepts optional additional inputs, which Snap! puts in a list; that list is named **args** inside the block. **Ask** has two nested **call** blocks. The inner one calls the object, i.e., the dispatch procedure. The dispatch procedure always takes exactly one input, namely the message. It reports a method, which may take any number of inputs; note that this is the situation in which we drop a list of values onto the arrowheads of a multiple input (in the outer **call** block). Note also that this is one of the rare cases in which we must unringify the inner **call** block, whose *value when called* gives the method.

C. Inheritance via Delegation

So, our objects now have local state variables and message passing. What about inheritance? We can provide that capability using the technique of *delegation*. Each instance of the child class contains an instance of the parent class, and simply passes on the messages it doesn't want to specialize:
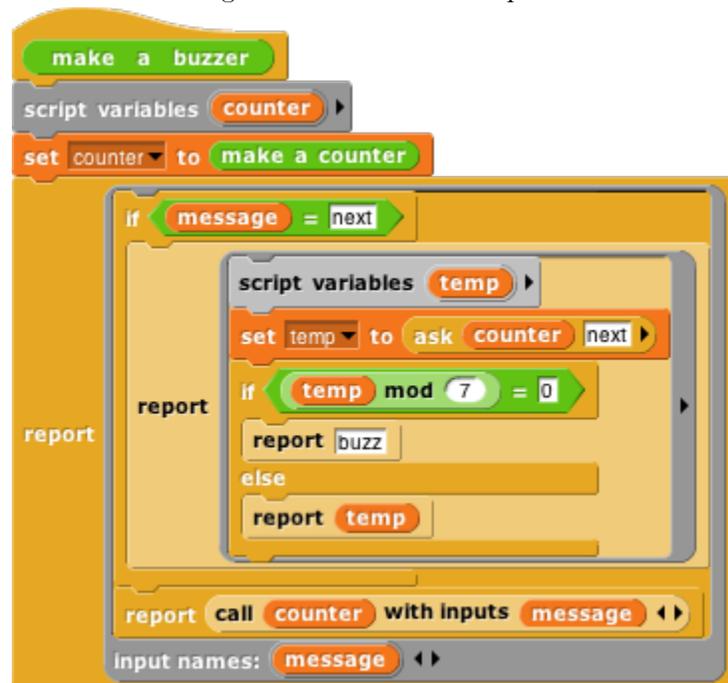
This script implements the **buzzer** class, which is a child of **counter**. Instead of having a count (a number) as a local state variable, each buzzer has a **counter** (an object) as a local state variable. The class specializes the **next** method, reporting what the counter reports unless that result is divisible by 7, in which case it reports "**buzz**." (Yeah, it should also check for a digit 7 in the number, but this code is complicated enough already.) If the message is anything other than **next**, though, such as **reset**, then the buzzer simply invokes its **counter**'s dispatch procedure. So the counter handles any message that the buzzer doesn't handle explicitly. (Note that in the non-**next** case we **call** the counter, not **ask** it something, because we want to report a method, not the value that the message reports.) So, if we **ask** a **buzzer** to **reset** to a value divisible by 7, it will end up reporting that number, not "**buzz**."

D. An Implementation of Prototyping OOP

In the class/instance system above, it is necessary to design the complete behavior of a class before you can make any instances of the class. This is okay for top-down design, but not great for experimentation. Here we sketch the implementation of a *prototyping* OOP system: You make an object, tinker with it, make clones of it, and keep tinkering. Any changes you make in the parent are inherited by its children. In effect, that first object is both the class and an instance of the class. In the implementation below, children share properties (methods and local variables) of their parent unless and until a child changes a property, at which point that child gets a private copy. (If a child wants to change something for its entire family, it must ask the parent to do it.)

Because we want to be able to create and delete properties dynamically, we won't use Snap! variables to hold an object's variables or methods. Instead, each object has two *tables,* called **methods** and **data**, each of which is an *association list:* a list of two-item lists, in which each of the latter contains a *key* and a corresponding *value.* We provide a lookup procedure to locate the key-value pair corresponding to a given key in a given table.

```
assoc (key) (alist :)
for (i) = (1) to (length of (alist))
  if <item (1▼) of (item (i) of (alist)) = (key)>
    report (item (i) of (alist))
report (list ▶)
```

```
found? (assoc-result :)
report <not <empty? (assoc-result)>>
```

```
assoc c (list (list a b ◀▶) (list c d ◀▶) (list e f ◀▶) (list c h ◀▶) ◀▶)
```

```
1  c  -
2  d  -
⊕ length: 2
```

```
assoc x (list (list a b ◀▶) (list c d ◀▶) (list e f ◀▶) (list c h ◀▶) ◀▶)
```

```
⊕ length: 0
```
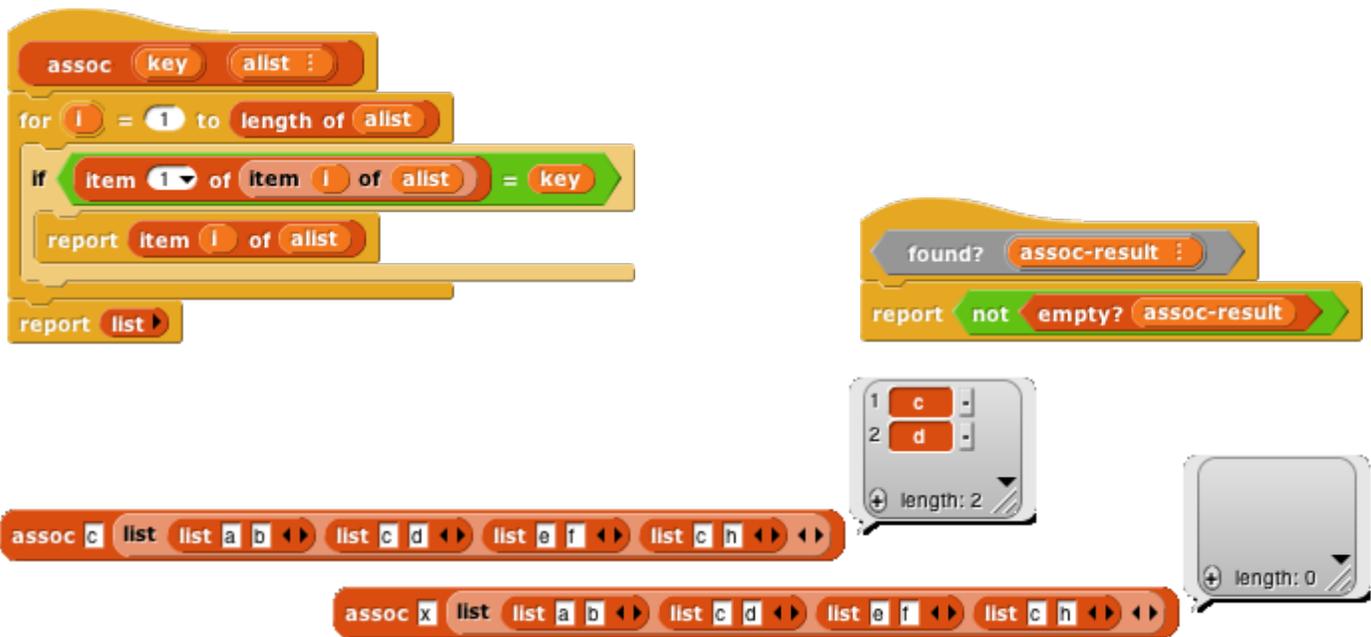
There are also commands to insert and delete entries:

```
insert (name) (value) into (table :)
script variables (kv-pair ▶)
set kv-pair▼ to (assoc (name) (table))
if <found? (kv-pair)>
  replace item (2▼) of (kv-pair) with (value)
else
  insert (list (name) (value) ◀▶) at (1▼) of (table)
```

```
delete (key) from (table :)
for (index) = (1) to (length of (table))
  if <item (1▼) of (item (index) of (table)) = (key)>
    delete (index) of (table)
    stop block
```
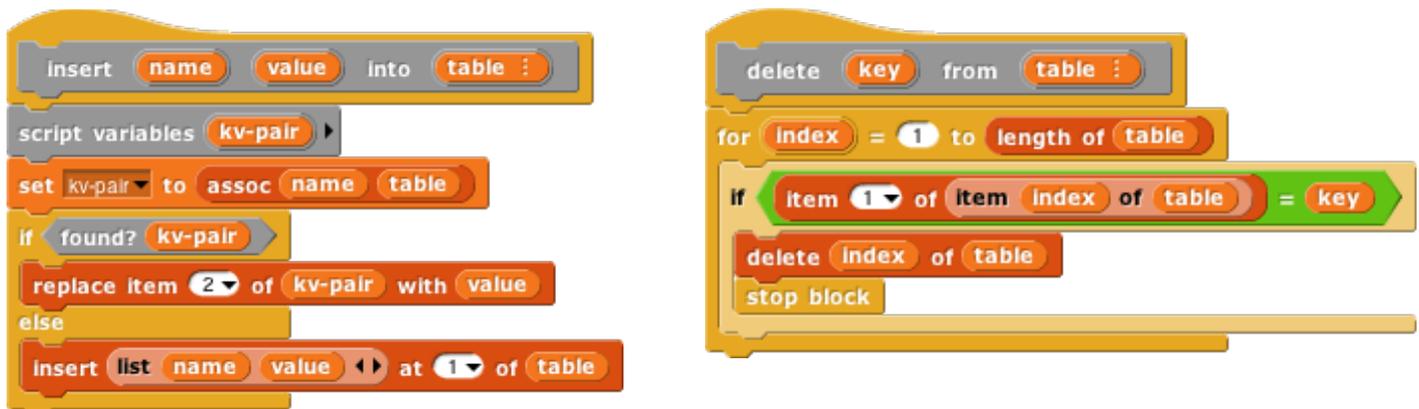
As in the class/instance version, an object is represented as a dispatch procedure that takes a message as its input and reports the corresponding method. When an object gets a message, it will first look for that keyword in its **methods** table. If it's found, the corresponding value is the method we want. If not, the object looks in its **data** table. If a value is found there, what the object returns is *not* that value, but rather a reporter method that, when called, will report the value. This means that what an object returns is *always* a method.

If the object has neither a method nor a datum with the desired name, but it does have a parent, then the parent (that is, the parent's dispatch procedure) is invoked with the message as its input. Eventually, either a match is found, or an object with no parent is found; the latter case is an error, meaning that the user has sent the object a message not in its repertoire.

Messages can take any number of inputs, as in the class/instance system, but in the prototyping version, every method automatically gets the object to which the message was originally sent as an extra first input. We must do this so that if a method is found in the parent (or grandparent, etc.) of the original recipient, and that method refers to a variable or method, it will use the child's variable or method if the child has its own version.

The **clone of** block below takes an object as its input and makes a child object. It should be considered as an internal part of the implementation; the preferred way to make a child of an object is to send that object a **clone** message.

Every object is created with predefined methods for **set**, **method**, **delete-var**, **delete-method**, and **clone**. It has one predefined variable, **parent**. Objects without a parent are created by calling **new object**:

As before, we provide procedures to call an object's dispatch procedure and then call the method. But in this version, we provide the desired object as the first method input. We provide one procedure for Command methods and one for Reporter methods:

The script below demonstrates how this prototyping system can be used to make counters. We start with one prototype counter, called **counter1**. We count this counter up a few times, then create a child **counter2** and give it its own **count** variable, but *not* its own **total** variable. The **next** method always sets **counter1**'s **total** variable, which therefore keeps count of the total number of times that *any* counter is incremented. Running this script should **[say]** and **(think)** the following lists:

[1 1] [2 2] [3 3] [4 4] (1 5) (2 6) (3 7) [5 8] [6 9] [7 10] [8 11]
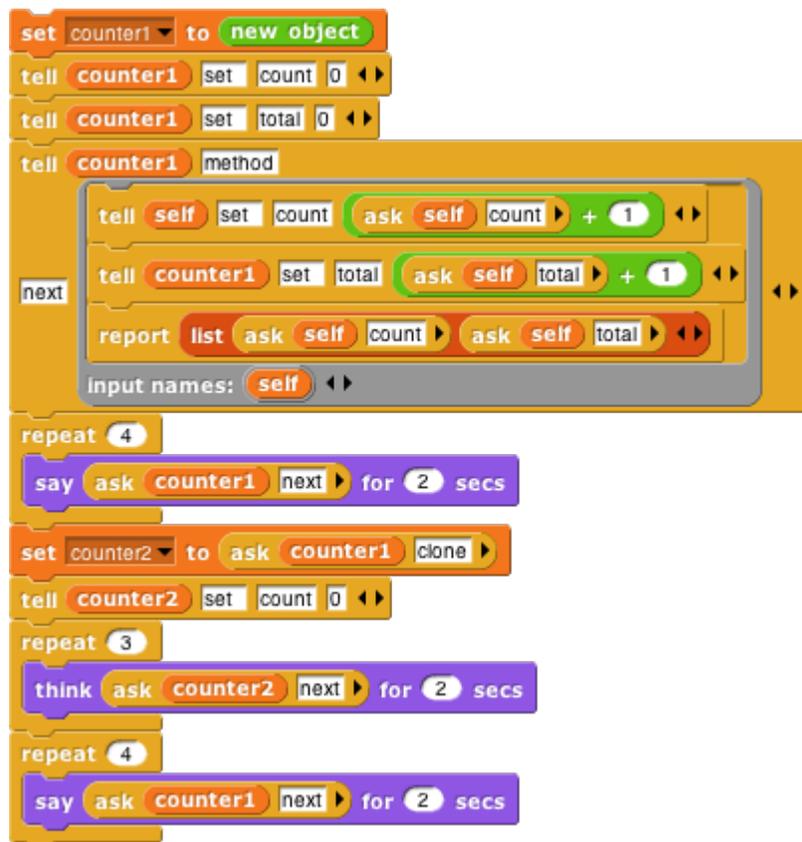
VIII. Continuations

Blocks are usually used within a script. The *continuation* of a block within a particular script is the part of the computation that remains to be completed after the block does its job. A continuation can be represented as a ringed script. Continuations are always part of the interpretation of any program in any language, but usually these continuations are implicit in the data structures of the language interpreter or compiler. Making continuations explicit is an advanced but versatile programming technique that allows users to create control structures such as nonlocal exit and multithreading.

In the simplest case, the continuation of a command block may just be the part of the script after the block. For example, in the script
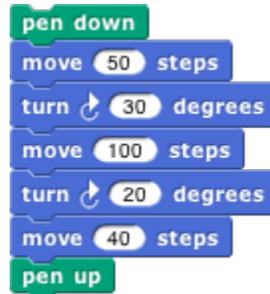
the continuation of the **move 100 steps** block is

But some situations are more complicated. For example, what is the continuation of **move 100 steps** in the following script?

That's a trick question; the **move** block is run four times, and it has a different continuation each time. The first time, its continuation is

Note that there is no **repeat 3** block in the actual script, but the continuation has to represent the fact that there are three more times through the loop to go. The fourth time, the continuation is just

What counts is not what's physically below the block in the script, but what computational work remains to be done.

When a block is used inside a custom block, its continuation may include parts of more than one script. For example, if we make a custom **square** block
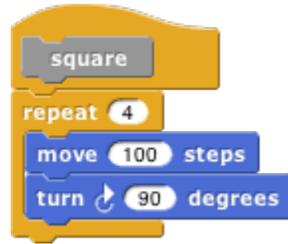
and then use that block in a script:

then the continuation of the first use of **move 100 steps** is

in which part comes from inside the **square** block and part comes from the call to **square**. Nevertheless, ordinarily when we *display* a continuation we show only the part within the current script.

The continuation of a command block, as we've seen, is a simple script with no input slots. But the continuation of a reporter block has to do something with the value reported by the block, so it takes that value as input. For example, in the script

the continuation of the **3+4** block is

Of course the name **result** in that picture is arbitrary; any name could be used, or no name at all by using the empty-slot notation for input substitution.

A. Continuation Passing Style

Like all programming languages, Snap! evaluates compositions of nested reporters from the inside out. For example, in the expression Snap! first adds 4 and 5, then multiplies 3 by that sum. This often means that the order in which the operations are done is backwards from the order in which they appear in the expression: When reading the above expression you say "times" before you say "plus." In English, instead of saying "three times four plus five," which actually makes the order of operations ambiguous, you could say, "take the sum of four and five, and then take the product of three and that sum." This sounds more awkward, but it has the virtue of putting the operations in the order in which they're actually performed.

That may seem like overkill in a simple expression, but suppose you're trying to convey the expression

factorial `3` × factorial (`2` + `2`) + `5`

to a friend over the phone. If you say "factorial of three times factorial of two plus two plus five" you might mean any of these:

factorial `3` × factorial (`2` + `2` + `5`)

factorial `3` × (factorial `2` + (`2` + `5`))

factorial `3` × (factorial `2`) + (`2` + `5`)

factorial `3` × (factorial `2` + `2`) + `5`

Wouldn't it be better to say, "Add two and two, take the factorial of that, add five to that, multiply three by that, and take the factorial of the result"? We can do a similar reordering of an expression if we first define versions of all the reporters that take their continuation as an explicit input. In the following picture, notice that the new blocks are *commands*, not reporters.

add `a` `b` `cont λ`
run `cont` with inputs (`a` + `b`)

subtract `a` `b` `cont λ`
run `cont` with inputs (`a` − `b`)

multiply `a` `b` `cont λ`
run `cont` with inputs (`a` × `b`)

equals? `a` `b` `cont λ`
run `cont` with inputs ⟨ `a` = `b` ⟩

factorial `n` `cont λ`
equals? `n` `0`
if `eqresult`
  run `cont` with inputs `1`
else
  subtract `n` `1`
    factorial `subresult`
      multiply `n` `factresult` `cont`   input names: `factresult`
    input names: `subresult`
  input names: `eqresult`

We can check that these blocks give the results we want:

factorial `5` say ▯

120

The original expression can now be represented as
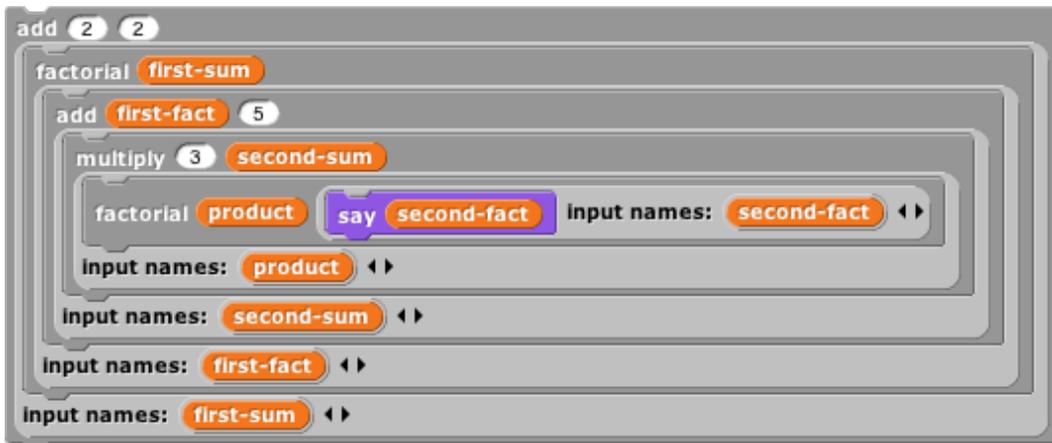
If you read this top to bottom, don't you get "Add two and two, take the factorial of that, add five to that, multiply three by that, and take the factorial of the result"? Just what we wanted! This way of working, in which every block is a command that takes a continuation as one of its inputs, is called *continuation-passing style (CPS)*. Okay, it looks horrible, but it has subtle virtues. One of them is that each script is just one block long (with the rest of the work buried in the continuation given to that one block), so each block doesn't have to remember what else to do — in the vocabulary of this chapter, the (implicit) continuation of each block is empty. Instead of the usual picture of recursion, with a bunch of little people all waiting for each other, with CPS what happens is that each little person hands off the problem to the next one and goes to the beach, so there's only one active little person at a time. In this example, we start with Alfred, an **add** specialist, who computes the value 4 and then hands off the rest of the problem to Francine, a **factorial** specialist. She computes the value 24, then hands the problem off to Anne, another **add** specialist, who computes 29. And so on, until finally Sam, a **say** specialist, says the value $2.107757298379527 \times 10^{132}$, which is a very large number!
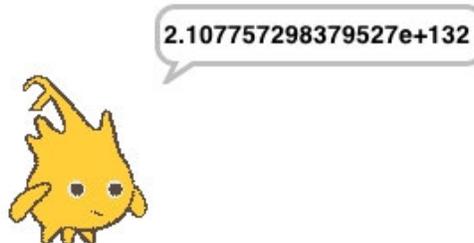
Go back to the definitions of these blocks. The ones, such as **add**, that correspond to primitive reporters are simple; they just call the reporter and then call their continuation with its result. But the definition of **factorial** is more interesting. It doesn't just call our original **factorial** reporter and send the result to its continuation. CPS is used inside **factorial** too! It says, "See if my input is zero. Send the (true or false) result to **if**. If the result is **true**, then call my continuation with the value 1. Otherwise, subtract 1 from my input. Send the result of that to (a recursive call to) **factorial**, with a continuation that multiplies the smaller number's factorial by my original input. Finally, call my continuation with the product." You can use CPS to unwind even the most complicated branched recursions.

By the way, I cheated a bit above. The **if/else** block should also use CPS; it should take one true/false input and *two continuations*. It will go to one or the other continuation depending on the value of its input. But in fact the C-shaped blocks (or E-shaped, like **if/else**) are really using CPS in the first place, because they implicitly wrap rings around the sub-scripts within their branches. See if you can make an explicitly CPS **if/else** block.

B. Call/Run w/Continuation

To use explicit continuation passing style, we had to define special versions of all the reporters, **add** and so on. Snap! provides a primitive mechanism for capturing continuations when we need to, without using continuation passing throughout a project.

Here's the classic example. We want to write a recursive block that takes a list of numbers as input, and reports the product of all the numbers:

But we can improve the efficiency of this block, in the case of a list that includes a zero; as soon as we see the zero, we know that the entire product is zero.

But this is not as efficient as it might seem. Consider, as an example, the list 1,2,3,0,4,5. We find the zero on the third recursive call (the fourth call altogether), as the first item of the sublist 0,4,5. What is the continuation of the **report 0** block? It's

Even though we already know that **result** is zero, we're going to do three unnecessary multiplications while unwinding the recursive calls.

We can improve upon this by capturing the continuation of the top-level call to **product**:

49

The block takes as its input a one-input script, as shown in the **product** example. It calls that script with *the continuation of the **call-with-continuation** block itself* as its input. In this case, that continuation is

reporting to whichever script called **product**. If the input list doesn't include a zero, then nothing is ever done with that continuation, and this version works just like the original **product**. But if the input list is 1,2,3,0,4,5, then three recursive calls are made, the zero is seen, and **product-helper** *runs the continuation,* with an input of 0. The continuation immediately reports that 0 to the caller of **product**, *without* unwinding all the recursive calls and without the unnecessary multiplications.

I could have written **product** a little more simply using a Reporter ring instead of a Command ring:

but it's customary to use a script to represent the input to **call w/continuation** because very often that input takes the form

so that the continuation is saved permanently and can be called from anywhere in the project. That's why the input slot in **call w/continuation** has a Command ring rather than a Reporter ring.

Nonlocal exit

Many programming languages have a **break** command that can be used inside a looping construct such as **repeat** to end the repetition early. Using first class continuations, we can generalize this mechanism to allow nonlocal exit even within a block called from inside a loop, or through several levels of nested loops:

The upvar **break** has as its value a continuation that can be called from anywhere in the program to jump immediately to whatever comes after the **catch** block in its script. Here's an example with two nested invocations of **catch**, with the upvar renamed in the outer one:
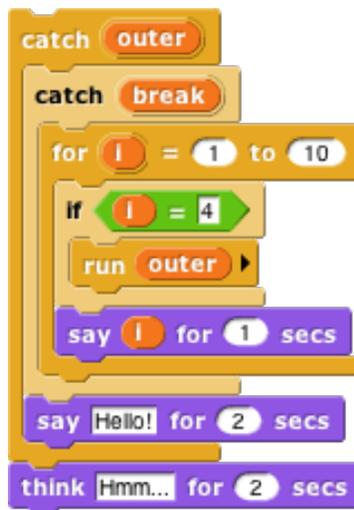
As shown, this will say 1, then 2, then 3, then exit both nested **catch**es and think "Hmm." If in the **run** block the variable **break** is used instead of **outer**, then the script will say 1, 2, 3, and "Hello!" before thinking "Hmm."

There are corresponding **catch** and **throw** blocks for reporters. The **catch** block is a reporter that takes an expression as input instead of a C-shaped slot. But the **throw** block is a command; it doesn't report a value to its own continuation, but instead reports a value (which it takes as an additional input, in addition to the **catch** tag) to *the corresponding **catch** block*'s continuation:

Without the **throw**, the inner **call** reports 5, the **+** block reports 8, so the **catch** block reports 8, and the **×** block reports 80. With the **throw**, the inner **call** doesn't report at all, and neither does the **+** block. The **throw** block's input of 20 becomes the value reported by the **catch** block, and the **×** block multiplies 10 and 20.

Creating a Thread System

Snap! can be running several scripts at once, within a single sprite and across many sprites. If you only have one computer, how can it do many things at once? The answer is that only one is actually running at any moment, but Snap! switches its attention from one script to another frequently. At the bottom of every looping block (**repeat**, **repeat until**, **forever**), there is an implicit "yield" command, which remembers where the current script is up to, and switches to some other script, each in turn. At the end of every script is an implicit "end thread" command (a *thread* is the technical term for the process of running a script), which switches to another script without remembering the old one.

Since this all happens automatically, there is generally no need for the user to think about threads. But, just to show that this, too, is not magic, here is an implementation of a simple thread system. It uses a global variable named **tasks** that initially contains an empty list. Each use of the C-shaped **thread** block adds a continuation (the ringed script) to the list. The **yield** block uses **run w/continuation** to create a continuation for a partly done thread, adds it to the task list, and then runs the first waiting task. The **end thread** block (which is automatically added at the end of every thread's script by the **thread** block) just runs the next waiting task.
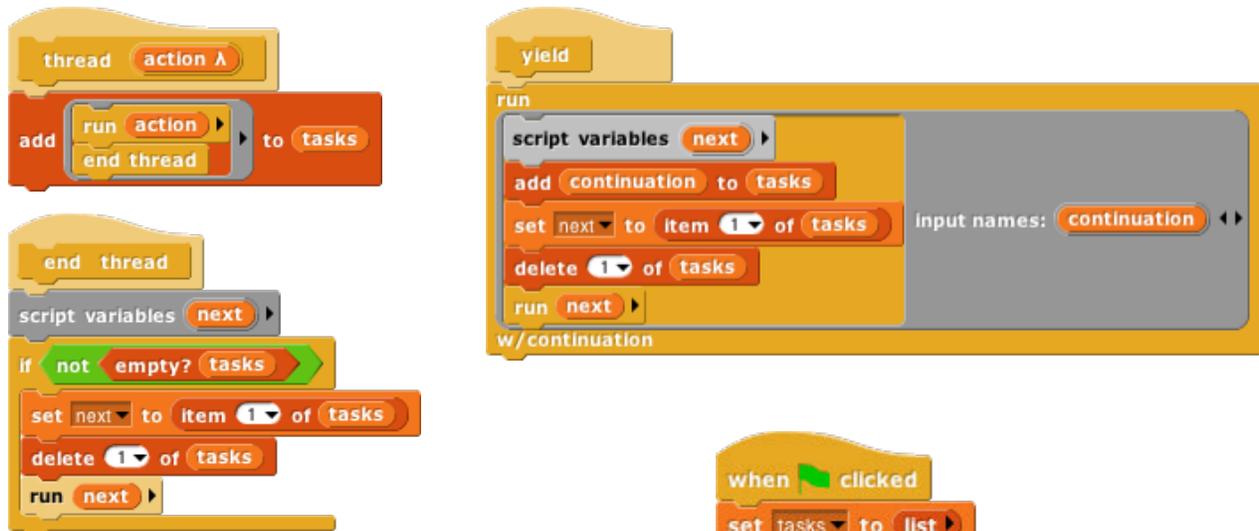
Here is a sample script using the thread system. One thread **say**s numbers; the other **say**s letters. The number thread yields after every prime number, while the letter thread yields after every vowel. So the sequence of speech balloons is 1,2,a,3,b,c,d,e,4,5,f,g,h,i,6,7,j,k,l,m,n,o,8,9,10,11, p,q,r,s,t,u,12,13,v,w,x,y,z,14,15,16,17,18,…30.
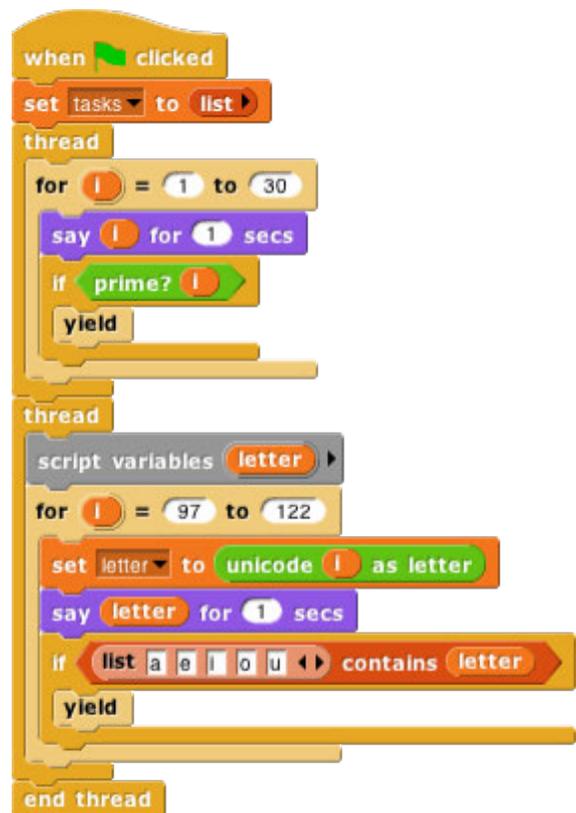
If we wanted this to behave exactly like Snap!'s own threads, we'd define new versions of **repeat** and so on that run **yield** after each repetition.

52

Index